

Netty®

权威指南

李林锋 / 著

Java高性能NIO通信首选框架

大数据时代构建高可用分布式系统利器

Netty: The Definitive Guide

□ □ □ □

Netty 5.0
Netty
Java I/O
Netty NIO
Netty
Netty

Java NIO 与 Java 的 Netty

□ □

[illegible]

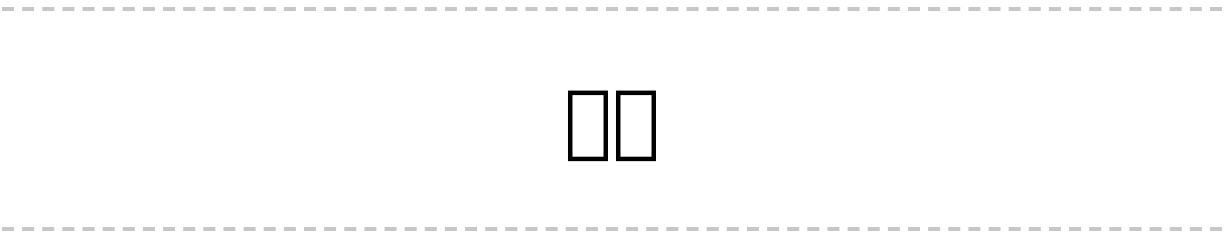
□□□□□□□ **CIP** □□□

Netty——2014.6

ISBN 978-7-121-23343-2

I ① N... II ① ... III ① JAVA... IV ① TP312-62

□□□□□□CIP□□□□□2014□□114646□



Tomcat Servlet I/O Java HTTP

NIOは2009年に登場したNIOのMinaという
Javaで書かれたライブラリが、
“”と記述されている2つの
10と記述されているNIOの
BUGは20~30個

2004JDK1.4NIO 1.010JSR 51JavaI/OJavaNIO

1CC++I/OJDKNIO

2JavaI/O

3C++JavaNIO

4J2EEI/OServletTomcatNIO

5NIONIO

6NIO

7NIO

NIOJavaNIONIONIODubboRocketMQAvroNettyNettyNetty

1

2

3

4

5

6

2014年，Netty 5.0 1
4000 NIO NIO
Netty NIO Netty
UserGuide Netty NIO
Netty NIO
Netty 2011 Netty
Mina 3 3
Netty Mina Netty
NIO Netty NIO
Netty

Netty
Netty
Netty

neu_lilinfeng@sina.com

Nettying

Nettying

Netty

Netty NIO

511



□□

[□□□_□□Java NIO](#)

[□1□_Java□I/O□□□□](#)

[1.1_ I/O□□□□](#)

[1.1.1_ Linux□□I/O□□□□](#)

[1.1.2_ I/O□□□□□□](#)

[1.2_ Java□I/O□□](#)

[1.3_ □□](#)

[□2□_ NIO□□](#)

[2.1_ □□□BIO□□](#)

[2.1.1_ BIO□□□□□](#)

[2.1.2_ □□□□□I/O□□□TimeServer□□□□](#)

[2.1.3_ □□□□□I/O□□□TimeClient□□□□](#)

[2.2_ □□□I/O□□](#)

[2.2.1_ □□□I/O□□□](#)

[2.2.2_ □□□□I/O□□□TimeServer□□□□](#)

[2.2.3_ □□□I/O□□□□](#)

[2.3_ NIO□□](#)

[2.3.1_ NIO□□□□](#)

[2.3.2_ NIO□□□□□□](#)

[2.3.3_ NIO□□□TimeServer□□□□](#)

[2.3.4_ NIO□□□□□□](#)

[2.3.5_ NIO□□□TimeClient□□□□](#)

2.4 AIO

2.4.1 AIOTimeServer

2.4.2 AIOTimeClient

2.4.3 AIO

2.5 4I/O

2.5.1

2.5.2 I/O

2.6 Netty

2.6.1 JavaNIO

2.6.2 Netty

2.7

Netty NIO

3 Netty

3.1 Netty

3.1.1 Netty

3.1.2 Netty

3.2 Netty

3.3 Netty

3.4

3.4.1

3.4.2

3.5

4 TCP/

4.1 TCP/

4.1.1 TCP/

4.1.2 TCP/

4.1.3

4.2 TCP

4.2.1 TimeServer

4.2.2 TimeClient

4.2.3

4.3 LineBasedFrameDecoderTCP

4.3.1 TCPTimeServer

4.3.2 TCPTimeClient

4.3.3 TCP

4.3.4 LineBasedFrameDecoderStringDecoder

4.4

5

5.1 DelimiterBasedFrameDecoder

5.1.1 DelimiterBasedFrameDecoder

5.1.2 DelimiterBasedFrameDecoder

5.1.3 DelimiterBasedFrameDecoder

5.2 FixedLengthFrameDecoder

5.2.1 FixedLengthFrameDecoder

5.2.2 telnetEchoServer

5.3

Netty

6

6.1 Java

6.1.1

6.1.2

6.1.3

6.2

6.2.1 GoogleProtobuf

[6.2.2 Facebook Thrift](#)

[6.2.3 JBoss Marshalling](#)

[6.3](#)

[7 Java](#)

[7.1 Netty Java](#)

[7.2 Java Netty](#)

[7.3](#)

[7.4](#)

[8 Google Protobuf](#)

[8.1 Protobuf](#)

[8.1.1 Protobuf](#)

[8.1.2 Protobuf](#)

[8.1.3 Protobuf](#)

[8.2 Netty Protobuf](#)

[8.2.1 Protobuf](#)

[8.2.2 Protobuf](#)

[8.2.3 Protobuf](#)

[8.3 Protobuf](#)

[8.4](#)

[9 JBoss Marshalling](#)

[9.1 Marshalling](#)

[9.2 Netty Marshalling](#)

[9.3 Netty Marshalling](#)

[9.4 Marshalling](#)

[9.5](#)

[Netty](#)

[10 HTTP](#)

[10.1 HTTP](#)

[10.1.1 HTTP URL](#)

[10.1.2 HTTP HttpRequest](#)

[10.1.3 HTTP HttpResponse](#)

[10.2 Netty HTTP](#)

[10.2.1 HTTP](#)

[10.2.2 HTTP](#)

[10.2.3 Netty HTTP](#)

[10.3 Netty HTTP+XML](#)

[10.3.1](#)

[10.3.2 HTTP+XML](#)

[10.3.3 XML JiBx](#)

[10.3.4 HTTP+XML](#)

[10.3.5 HTTP+XML](#)

[10.3.6](#)

[10.4](#)

[11 WebSocket](#)

[11.1 HTTP](#)

[11.2 WebSocket](#)

[11.2.1 WebSocket](#)

[11.2.2 WebSocket](#)

[11.2.3 WebSocket](#)

[11.2.4 WebSocket](#)

[11.3 Netty WebSocket](#)

[11.3.1 WebSocket](#)

[11.3.2 WebSocket](#)

[11.3.3 WebSocket](#)

11.4 [IO](#)

12 [UDP](#)

12.1 [UDP](#)

12.2 [UDP](#)

12.3 [UDP](#)

12.4 [UDP](#)

12.5 [IO](#)

13 [IO](#)

13.1 [IO](#)

13.1.1 [IO](#)

13.1.2 [IO](#)

13.1.3 [IO](#)

13.1.4 [FileChannel](#)

13.2 [Netty](#)

13.3 [Netty](#)

13.4 [IO](#)

14 [IO](#)

14.1 [IO](#)

14.2 [Netty](#)

14.2.1 [IO](#)

14.2.2 [IO](#)

14.2.3 [IO](#)

14.2.4 [IO](#)

14.2.5 [Netty](#)

14.2.6 [Netty](#)

14.2.7 [IO](#)

14.2.8 [IO](#)

[14.2.9 `ByteBuf`](#)

[14.2.10 `ByteBuf`](#)

[14.2.11 `ByteBuf`](#)

[14.3 `Netty`](#)

[14.3.1 `ByteBuf`](#)

[14.3.2 `ByteBuf`](#)

[14.3.3 `ByteBuf`](#)

[14.3.4 `ByteBuf`](#)

[14.3.5 `ByteBuf`](#)

[14.3.6 `ByteBuf`](#)

[14.3.7 `ByteBuf`](#)

[14.4 `ByteBuf`](#)

[14.4.1 `ByteBuf`](#)

[14.4.2 `ByteBuf`](#)

[14.4.3 `ByteBuf`](#)

[14.5 `ByteBuf`](#)

[`ByteBuf` `Netty`](#)

[15 `ByteBuf`](#)

[15.1 `ByteBuf`](#)

[15.1.1 `ByteBuf`](#)

[15.1.2 `ByteBuf`](#)

[15.2 `ByteBuf`](#)

[15.2.1 `ByteBuf`](#)

[15.2.2 `AbstractByteBuf`](#)

[15.2.3 `AbstractReferenceCountedByteBuf`](#)

[15.2.4 `UnpooledHeapByteBuf`](#)

[15.2.5 `PooledByteBuf`](#)

[15.2.6 PooledDirectByteBuffer](#)

[15.3 ByteBuffer](#)

[15.3.1 ByteBufferHolder](#)

[15.3.2 ByteBufferAllocator](#)

[15.3.3 CompositeByteBuffer](#)

[15.3.4 ByteBufferUtil](#)

[15.4](#)

[16 ChannelUnsafe](#)

[16.1 Channel](#)

[16.1.1 Channel](#)

[16.1.2 Channel](#)

[16.2 Channel](#)

[16.2.1 Channel](#)

[16.2.2 AbstractChannel](#)

[16.2.3 AbstractNioChannel](#)

[16.2.4 AbstractNioByteChannel](#)

[16.2.5 AbstractNioMessageChannel](#)

[16.2.6 AbstractNioMessageServerChannel](#)

[16.2.7 NioServerSocketChannel](#)

[16.2.8 NioSocketChannel](#)

[16.3 Unsafe](#)

[16.4 Unsafe](#)

[16.4.1 Unsafe](#)

[16.4.2 AbstractUnsafe](#)

[16.4.3 AbstractNioUnsafe](#)

[16.4.4 NioByteUnsafe](#)

[16.5](#)

[17 ChannelPipelineChannelHandler](#)

[17.1 ChannelPipeline](#)

[17.1.1 ChannelPipeline](#)

[17.1.2](#)

[17.1.3](#)

[17.1.4 ChannelPipeline](#)

[17.2 ChannelPipeline](#)

[17.2.1 ChannelPipeline](#)

[17.2.2 ChannelPipelineChannelHandler](#)

[17.2.3 ChannelPipelineinbound](#)

[17.2.4 ChannelPipelineoutbound](#)

[17.3 ChannelHandler](#)

[17.3.1 ChannelHandlerAdapter](#)

[17.3.2 ByteToMessageDecoder](#)

[17.3.3 MessageToMessageDecoder](#)

[17.3.4 LengthFieldBasedFrameDecoder](#)

[17.3.5 MessageToByteEncoder](#)

[17.3.6 MessageToMessageEncoder](#)

[17.3.7 LengthFieldPrepender](#)

[17.4 ChannelHandler](#)

[17.4.1 ChannelHandler](#)

[17.4.2 ByteToMessageDecoder](#)

[17.4.3 MessageToMessageDecoder](#)

[17.4.4 LengthFieldBasedFrameDecoder](#)

[17.4.5 MessageToByteEncoder](#)

[17.4.6 MessageToMessageEncoder](#)

[17.4.7 LengthFieldPrepender](#)

17.5 □□

18 EventLoopEventLoopGroup

18.1 Netty

18.1.1 Reactor□□□□□

18.1.2 Reactor□□□□□

18.1.3 Reactor

18.1.4 Netty

18.1.5 □□□□

18.2 NioEventLoop

18.2.1 NioEventLoop

18.2.2 NioEventLoop

18.2.3 NioEventLoop

18.3 □□

□19□ Future□Promise

19.1 Future

19.2 ChannelFuture□□□□

19.3 Promise□□□□

19.4 Promise

19.4.1 Promise

19.4.2 DefaultPromise

19.5 □□

Netty

20_ Java_ByteBuffer_Netty_

20.1 Java

20.1.1 浮点数的表示

20.1.2 Java

20.2 Netty

[20.2.1 多线程环境下的原子性](#)

[20.2.2 内存屏障](#)

[20.2.3 volatile关键字](#)

[20.2.4 CAS操作](#)

[20.2.5 内存模型](#)

[20.2.6 内存一致性](#)

[20.2.7 顺序一致性](#)

[20.2.8 内存模型与处理器](#)

[20.3 小结](#)

[第21章 Netty入门](#)

[21.1 Netty简介](#)

[21.1.1 Reactor模型](#)

[21.1.2 什么是ChannelPipeline](#)

[21.1.3 什么是Service ChannelHandler](#)

[21.2 快速入门](#)

[21.2.1 安装](#)

[21.2.2 配置](#)

[21.2.3 启动](#)

[21.2.4 测试](#)

[21.3 小结](#)

[第22章 Netty进阶](#)

[22.1 Netty与数据库](#)

[22.1.1 数据库连接池](#)

[22.1.2 数据库连接池Dubbo](#)

[22.1.3 Dubbo与数据库](#)

[22.1.4 Netty与Dubbo](#)

[22.1.5 Dubbo与Netty](#)

[22.2 NettyByteBuffer](#)

[22.3 NettyChannel](#)

[22.3.1 ChannelHandler](#)

[22.3.2 NettyChannelHandler](#)

[22.4 Channel](#)

[23 Netty](#)

[23.1 Channel](#)

[23.2 Channel](#)

[23.3 Channel](#)

[23.4 Road Map](#)

[23.5 Channel](#)

[Netty](#)

Java NIO

1. Java I/O

2. NIO

第1章 Java I/O 概述

Java 由 Sun Microsystems 公司于 1995 年发明并实现。Java 语言最初是作为网络上的分布式计算而设计的。Java 语言在 8.5 版本中引入了对 I/O 的支持。

Java 语言最初是作为网络上的分布式计算而设计的。Java 语言最初是作为网络上的分布式计算而设计的。Java 语言最初是作为网络上的分布式计算而设计的。

Java 语言最初是作为网络上的分布式计算而设计的。Java 语言最初是作为网络上的分布式计算而设计的。Java 语言最初是作为网络上的分布式计算而设计的。Java 语言最初是作为网络上的分布式计算而设计的。

Java 语言最初是作为网络上的分布式计算而设计的。Java 语言最初是作为网络上的分布式计算而设计的。Java 语言最初是作为网络上的分布式计算而设计的。Java 语言最初是作为网络上的分布式计算而设计的。

Java 语言最初是作为网络上的分布式计算而设计的。

- I/O 概述
- Java I/O 概述

1.1 I/O 概述

Java 1.4 版本中引入了对 I/O 的支持。Java 1.4 版本中引入了对 I/O 的支持。Java 1.4 版本中引入了对 I/O 的支持。

- 了解Linux I/O模型
- 了解C++ Channel模型
- 了解Linux I/O模型BIO模型
- 了解Linux I/O模型NIO模型

Java I/O模型与C++ I/O模型

1.1.1 Linux I/O模型

Linux I/O模型分为三类：阻塞I/O、非阻塞I/O、异步I/O。阻塞I/O是指，当进程发起I/O请求时，如果数据不在缓存中，进程就会阻塞，直到数据到达缓存为止。非阻塞I/O是指，当进程发起I/O请求时，如果数据不在缓存中，进程不会阻塞，而是立即返回，等待数据到达缓存。异步I/O是指，当进程发起I/O请求时，如果数据不在缓存中，进程不会阻塞，而是立即返回，等待数据到达缓存。file descriptor fd、socket、socketfd、socket等。

UNIX I/O模型分为三类：阻塞I/O、非阻塞I/O、异步I/O。

1. 阻塞I/O：当进程发起I/O请求时，如果数据不在缓存中，进程就会阻塞，直到数据到达缓存为止。recvfrom()函数用于从socket接收数据，如果数据不在缓存中，进程就会阻塞，直到数据到达缓存为止。recvfrom()函数返回的数据长度，如果数据不在缓存中，进程就会阻塞，直到数据到达缓存为止。recvfrom()函数返回的数据长度，如果数据不在缓存中，进程就会阻塞，直到数据到达缓存为止。1-1

1-1 I/O模型

2. 非阻塞I/O：当进程发起I/O请求时，如果数据不在缓存中，进程不会阻塞，而是立即返回，等待数据到达缓存。recvfrom()函数用于从socket接收数据，如果数据不在缓存中，进程不会阻塞，而是立即返回，等待数据到达缓存。recvfrom()函数返回的数据长度，如果数据不在缓存中，进程不会阻塞，而是立即返回，等待数据到达缓存。recvfrom()函数返回的数据长度，如果数据不在缓存中，进程不会阻塞，而是立即返回，等待数据到达缓存。1-2

1-2 I/O模型

3 I/O Linux select/poll fd select
poll select select/poll fd
select/poll fd fd Linux
epoll epoll fd
rollback 1-3

□1-3 I/O□□□□

```

4 I/O I/O sigaction
SIGIO recvfrom
1-4

```

□1-4 □□□□I/O□□

5 I/O
I/O
I/O I/O 1-5

□1-5 □□I/O□□

UNIX
API
Java
I/O
Java
Java NIO

```

    [ ]/[O]Java NIO[ ]Selector[ ]
    [ ]epoll[ ]

```

1.1.2 I/O多路复用

I/O多路复用是指通过一个系统调用，同时监视多个I/O操作，一旦某个I/O操作就绪，就立即通知应用程序。I/O多路复用技术包括select、poll、epoll等。select是Linux中最基本的I/O多路复用技术，它通过一个select系统调用，可以同时监视多个I/O操作。select的缺点是效率低，因为它需要遍历所有文件描述符。poll和epoll是更高效的多路复用技术，它们通过不同的数据结构来管理文件描述符。

- 阻塞I/O：应用程序在等待I/O操作完成时，会一直阻塞在那里，直到操作完成。
- 非阻塞I/O：应用程序在等待I/O操作完成时，不会阻塞，可以继续执行其他操作。

I/O多路复用技术包括select、pselect、poll、epoll等。Linux中，select是最基本的I/O多路复用技术，它通过一个select系统调用，可以同时监视多个I/O操作。select的缺点是效率低，因为它需要遍历所有文件描述符。poll和epoll是更高效的多路复用技术，它们通过不同的数据结构来管理文件描述符。epoll是Linux中最高效的多路复用技术，它通过一个epoll系统调用，可以同时监视多个I/O操作。

1. 通过socket创建FD，FD是文件描述符，它用于标识I/O操作。

select系统调用可以同时监视多个FD，FD的数量不能超过FD_SETSIZE，通常是1024。TCP连接的数量不能超过FD的数量。Apache是一个高性能的Web服务器，它使用了epoll技术来实现I/O多路复用。Linux中，epoll是最基本的I/O多路复用技术，它通过一个epoll系统调用，可以同时监视多个I/O操作。Java中，Socket是用于网络通信的类，它使用了select技术来实现I/O多路复用。epoll是Linux中最高效的多路复用技术，它通过一个epoll系统调用，可以同时监视多个I/O操作。FD的数量不能超过1024，1GB的内存空间。cat /proc/sys/fs/file-max可以查看系统文件描述符的最大数量。

2 I/O FD

select/poll socket
socket “” select/poll
epoll “” socket -
epoll fd callback “” socket
callback idle socket epoll AIO
epoll select benchmark socket -
LAN epoll select/poll epoll_ctl
idle connections WAN epoll
select/poll

3 mmap

select poll epoll FD mmap

4 epoll API

```

00000000  epoll00000000000000000000000000000000  epoll00000000

```

```

select/poll epoll epoll Linux
freeBSD kqueue dev/poll Solaris
kqueue freebsd kernel
select/poll signal kqueue
/dev/poll Solaris API Kernel
/dev/poll fd_set pollfd
ioctl select /dev/poll

```

1.2 Java I/O BIO
 NIO Java
 NIO NIO Web Java C
 C++

1.2 Java I/O

JDK 1.4
 Java NIO
 Java Socket
 BIO
 C
 C++
 I/O
 AIO
 Java BIO

Java BIO Java I/O
JDK1.4 NIO Java I/O

Java I/O

`JDK1.0``JDK1.3``java``I/O``UNIX``Pipe``Channel``Buffer``Selector``2002``JDK1.4``NIO``JSR-51``JDK``java.nio``API`

- `ByteBuffer`
- `Pipe`
- `Channel` `ServerSocketChannel` `SocketChannel`

- 阻塞IO模型
- 非阻塞IO模型
- 多路复用IO模型
- 异步IO模型

Java NIO 是 Java 7 中引入的，它提供了对底层操作系统的更直接访问，以及更高效的 I/O 操作。NIO 2.0 是 NIO 的升级版，它提供了更多的 API 和更好的性能。

- 阻塞IO模型
- API 简单，易于使用
- 阻塞IO模型
- 阻塞IO模型

2011年7月28日，JDK 1.7 发布，其中包含 NIO 2.0。NIO 2.0 是 NIO 的升级版，它提供了更多的 API 和更好的性能。

- 阻塞IO模型
- API 简单，易于使用
- 阻塞IO模型
- 阻塞IO模型

1.3 阻塞IO

阻塞IO模型是传统的IO模型，它在等待数据到达时会阻塞线程。在阻塞IO模型中，线程会一直等待，直到数据到达为止。这种模型在单线程应用中非常常见，但在多线程应用中，它会导致性能下降，因为一个线程的阻塞会影响其他线程的执行。

图2-1展示了基于BIO的服务器模型。在这种模型中，服务器端有一个或多个线程，它们不断地调用`Accept`方法来接收客户端的连接。一旦接收到连接，服务器就会创建一个新的线程来处理这个连接。这个新线程会调用`Socket`方法来获取客户端的套接字，然后调用`InputStream`和`OutputStream`方法来读取和写入数据。当处理完一个连接后，这个线程就会结束。服务器端的主线程会继续调用`Accept`方法来接收新的连接。

图2-1 基于BIO的服务器模型

图2-1展示了基于BIO的服务器模型。在这种模型中，服务器端有一个或多个线程，它们不断地调用`Accept`方法来接收客户端的连接。一旦接收到连接，服务器就会创建一个新的线程来处理这个连接。这个新线程会调用`Socket`方法来获取客户端的套接字，然后调用`InputStream`和`OutputStream`方法来读取和写入数据。当处理完一个连接后，这个线程就会结束。服务器端的主线程会继续调用`Accept`方法来接收新的连接。

图2-1展示了基于BIO的服务器模型。在这种模型中，服务器端有一个或多个线程，它们不断地调用`Accept`方法来接收客户端的连接。一旦接收到连接，服务器就会创建一个新的线程来处理这个连接。这个新线程会调用`Socket`方法来获取客户端的套接字，然后调用`InputStream`和`OutputStream`方法来读取和写入数据。当处理完一个连接后，这个线程就会结束。服务器端的主线程会继续调用`Accept`方法来接收新的连接。

2.1.2 基于BIO的TimeServer

图2-1 基于BIO的TimeServer

图2-1 基于BIO的TimeServer

```
1. package com.phei.netty.bio;
2. import java.io.IOException;
3. import java.net.ServerSocket;
4. import java.net.Socket;
5. /**
6.  * @author lilinfeng
7.  * @date 2014-2-14
8.  * @version 1.0
```

```

9.      */
10.     public class TimeServer {
11.
12.         /**
13.          * @param args
14.          * @throws IOException
15.          */
16.         public static void main(String[] args) throws
IOException {
17.             int port = 8080;
18.             if (args != null && args.length > 0) {
19.
20.                 try {
21.                     port = Integer.valueOf(args[0]);
22.                 } catch (NumberFormatException e) {
23.                     // 错误
24.                 }
25.
26.             }
27.             ServerSocket server = null;
28.             try {
29.                 server = new ServerSocket(port);
30.                 System.out.println("The time server is start
in port : " + port);
31.                 Socket socket = null;
32.                 while (true) {
33.                     socket = server.accept();

```

```
34.                                     new Thread(new
TimeServerHandler(socket)).start();
```

```
35.         }
36.     } finally {
37.         if (server != null) {
38.             System.out.println("The time server close");
39.             server.close();
40.             server = null;
41.         }
42.     }
43. }
44. }
```

TimeServer 8080 29
ServerSocket 32 35
ServerSocket accept
TimeServer JvisualVM accept
2-2


```

32.            this.socket.getInputStream()));
33.            out = new
PrintWriter(this.socket.getOutputStream(), true);
34.            String currentTime = null;
35.            String body = null;
36.            while (true) {
37.                body = in.readLine();
38.                if (body == null)
39.                    break;
40.                System.out.println("The time server receive
order : " + body);
41.                currentTime = "QUERY TIME
ORDER".equalsIgnoreCase(body) ? new java.util.Date(
42.                    System.currentTimeMillis()).toString() :
"BAD ORDER";
43.                out.println(currentTime);
44.            }
45.
46.        } catch (Exception e) {
47.            if (in != null) {
48.                try {
49.                    in.close();
50.                } catch (IOException e1) {
51.                    e1.printStackTrace();
52.                }
53.            }
54.            if (out != null) {

```

```

55.         out.close();
56.         out = null;
57.     }
58.     if (this.socket != null) {
59.         try {
60.             this.socket.close();
61.         } catch (IOException e1) {
62.             e1.printStackTrace();
63.         }
64.         this.socket = null;
65.     }
66. }
67. }
68. }

```

37. `BufferedReader` 是否为 `null`。
 如果不是 `null`，则从 `BufferedReader` 中读取一行，并打印到控制台。
 如果为 `null`，则打印 `"QUERY TIME ORDER"`。
 47. `PrintWriter` 的 `println` 方法。
 64. `Socket` 的 `close` 方法。

以上代码中，我们使用了 `IOException` 异常。
 在 Java 中，异常分为两大类：编译时异常和运行时异常。

2.1.3 使用 `I/O` 的 `TimeClient`

Socket"QUERY TIME ORDER"

2-3 I/O TimeClient

```
13. public class TimeClient {
14.
15.     /**
16.      * @param args
17.      */
18.     public static void main(String[] args) {
19.         int port = 8080;
20.         if (args != null && args.length > 0) {
21.             try {
22.                 port = Integer.valueOf(args[0]);
23.             } catch (NumberFormatException e) {
24.                 //
25.             }
26.         }
27.         Socket socket = null;
28.         BufferedReader in = null;
29.         PrintWriter out = null;
30.         try {
31.             socket = new Socket("127.0.0.1", port);
32.             in = new BufferedReader(new
InputStreamReader(
33.                 socket.getInputStream()));
```



```

34.                                     out    =    new
PrintWriter(socket.getOutputStream(), true);

35.                                     out.println("QUERY TIME ORDER");

36.                                     System.out.println("Send order 2 server
succeed.");

37.                                     String resp = in.readLine();
38.                                     System.out.println("Now is : " + resp);
39.     } catch (Exception e) {
40.         //□□□□□
41.     } finally {
42.         if (out != null) {
43.             out.close();
44.             out = null;
45.         }
46.
47.         if (in != null) {
48.             try {
49.                 in.close();
50.             } catch (IOException e) {
51.                 e.printStackTrace();
52.             }
53.             in = null;
54.         }
55.         if (socket != null) {
56.             try {
57.                 socket.close();
58.             } catch (IOException e) {

```

```

59.             e.printStackTrace();
60.         }
61.         socket = null;
62.     }
63. }
64. }
65. }

```

35. `PrintWriter` `"QUERY TIME ORDER"` `BufferedReader` `readLine`

2-3

2-3

2-3

2-4

2-4

BIO

1 N

2.2 网络I/O

网络I/O是指通过网络进行的数据传输。在Java中，网络I/O可以分为两种类型：阻塞I/O和非阻塞I/O。阻塞I/O是指在进行网络I/O操作时，程序会一直等待，直到操作完成为止。而非阻塞I/O则是指在进行网络I/O操作时，程序不会一直等待，而是可以继续执行其他操作，直到操作完成后再返回结果。

在Java中，网络I/O操作通常是通过Socket和ServerSocket类来实现的。Socket类用于创建客户端，而ServerSocket类用于创建服务器。通过这两个类，可以实现网络I/O操作。

2.2.1 网络I/O

网络I/O操作通常涉及到数据的发送和接收。在Java中，网络I/O操作通常是通过InputStream和OutputStream类来实现的。

图2-5 网络I/O操作流程图

在Java中，网络I/O操作通常是通过Socket和ServerSocket类来实现的。Socket类用于创建客户端，而ServerSocket类用于创建服务器。通过这两个类，可以实现网络I/O操作。

在Java中，网络I/O操作通常是通过Socket和ServerSocket类来实现的。Socket类用于创建客户端，而ServerSocket类用于创建服务器。通过这两个类，可以实现网络I/O操作。

2.2.2 网络I/O TimeServer

网络I/O操作通常涉及到数据的发送和接收。

图2-4 网络I/O TimeServer

```

13.    public class TimeServer {
14.
15.        /**
16.         * @param args
17.         * @throws IOException
18.         */
19.        public static void main(String[] args) throws
IOException {
20.            int port = 8080;
21.            if (args != null && args.length > 0) {
22.                try {
23.                    port = Integer.valueOf(args[0]);
24.                } catch (NumberFormatException e) {
25.                    // 端口错误
26.                }
27.            }
28.            ServerSocket server = null;
29.            try {
30.                server = new ServerSocket(port);
31.                System.out.println("The time server is start
in port : " + port);
32.                Socket socket = null;
33.                TimeServerHandlerExecutePool singleExecutor =
new TimeServerHandlerExecutePool(

```

```
34.                50, 10000);//
```

```
    I/O
```

```
35.                while (true) {
```

```
36.                socket = server.accept();
```

```
37.                                singleExecutor.execute(new  
TimeServerHandler(socket));
```

```
38.                }
```

```
39.            } finally {
```

```
40.                if (server != null) {
```

```
41.                    System.out.println("The time server close");
```

```
42.                    server.close();
```

```
43.                    server = null;
```

```
44.                }
```

```
45.            }
```

```
46.        }
```

```
47.    }
```


is thrown.

*

* <p> If the length of `b` is zero, then no bytes are read and

* `0` is returned; otherwise, there is an attempt to read at

* least one byte. If no byte is available because the stream is at the

* end of the file, the value `-1` is returned; otherwise, at

* least one byte is read and stored into `b`.

*

* <p> The first byte read is stored into element `b[0]`, the

* next one into `b[1]`, and so on. The number of bytes read is,

* at most, equal to the length of `b`. Let *k* be the

* number of bytes actually read; these bytes will be stored in elements

* `b[0]` through `b[k-1]`,

* leaving elements `b[k]` through

* `b[b.length-1]` unaffected.

*

* **@param**

b the buffer into which the data is read.

* **@return**

the total number of bytes read into the buffer, or
* `-1` if there is no more data
because the end of

* the stream has been reached.

* **@exception**

`IOException` If the first byte cannot be read for any reason
 * other than the end of the file, if the input stream
has been closed, or
 * if some other I/O error occurs.
 * *@exception*

`NullPointerException` if `b` is `null`
</code>.
 */
 public int

`read(byte`

`b[]) throws`

```
IIOException {  
    return
```

```
read(b, 0, b.length);  
}
```

API Socket
API

-
-
- I/O

60s I/O 60s

JDK I/O API

2-7 Java OutputStream

```
public void write(byte b[]) throws IOException
```

*Writes an array of bytes. This method will block until the bytes are *actually written.

Parameters:

b - the data to be written

Throws :

IOException

If an I/O error has occurred.

```

    OutputStream write(
        // TCP/IP
        // TCP
        // TCP window size
        // 0
        // Keep-Alive
        // TCP
        // I/O
        write(
            // TCP window size
            // 0
            // I/O

```

API
I/O I/O

I/O I/O I/O

1 60s 10ms

2 I/O
60s

3 I/O

4

5 Acceptor

6

□□□□□□□□□□NIO□□□□□□

2.3 NIO□□

1. NIO 2. New I/O 3. New I/O 4. Java I/O 5. I/O 6. Non-block I/O 7. I/O 8. NIO 9. I/O 10. NIO 11. I/O

```

    Socket
    ServerSocket
    NIO
    SocketChannel
    ServerSocketChannel
    I/O
    NIO

```

NIO

NIO

NIO API

NIO 2.3

2.3.1 NIO□□□□

1. NIO 1.4 版本中，NIO 的 I/O 操作是阻塞的。
 2. Java 7 中，NIO 的 I/O 操作是非阻塞的。
 3. NIO 的 I/O 操作是非阻塞的。
 4. NIO 的 I/O 操作是非阻塞的。

10000 Buffer

```

    ByteBuffer[] BufferPoolBufferPool
    NIOByteBufferI/OI/O
    Stream

```

`NIO`

`NIO`

```

ByteBuffer
limit

```

```

ByteBuffer[] ByteBuffer[] byte[]
ByteBuffer[] ByteBuffer[] Java[] Boolean[]
[]

```

- ByteBuffer
- CharBuffer

- ShortBuffer[]
- IntBuffer[]
- LongBuffer[]
- FloatBuffer[]
- DoubleBuffer[]

□□□□□□□□□□2-8□□□

□2-8 Buffer□□□□□

```

    Buffer Buffer ByteBuffer Buffer
    ByteBuffer I/O ByteBuffer
    ByteBuffer

```

2 Channel

```
Channel[]
Channel[]
InputStream[]OutputStream[]
```

```

ChannelAPI
UNIX

```

Channel 2-9

□2-9 Channel□□□□□□

Channel

Channel

SelectableChannel 和 FileChannel

ServerSocketChannel 和 SocketChannel 是 SelectableChannel 的子类

3. 选择器 Selector

Selector 是 Java NIO 中的一个类，它用于监视多个 Channel 是否就绪。Selector 是 NIO 中的一个核心组件，它允许一个线程监视多个 Channel。Channel 可以是 TCP 连接、文件通道等。Selector 通过 Channel 的 SelectionKey 来监视 Channel 的 I/O 状态。

Selector 的 Channel 是 JDK 的 epoll() 方法。select 方法最多只能监视 1024/2048 个 Channel。Selector 的 Channel 是 JDK 的 epoll() 方法。

NIO 的 Channel 是 NIO 的 Channel。NIO 的 Channel 是 NIO 的 Channel。

2.3.2 NIO 的 Channel

NIO 的 Channel 是 2-10 页

2-10 NIO 的 Channel

NIO 的 Channel 是 NIO 的 Channel。NIO 的 Channel 是 NIO 的 Channel。

创建ServerSocketChannel并打开它

```
ServerSocketChannel acceptorSvr =  
ServerSocketChannel.open();
```

创建并打开ServerSocketChannel

```
acceptorSvr.socket().bind(new  
InetSocketAddress(InetAddress.getByName("IP"), port));  
acceptorSvr.configureBlocking(false);
```

创建Reactor并打开它

```
Selector selector = Selector.open();  
New Thread(new ReactorTask()).start();
```

创建ServerSocketChannel并打开Reactor并打开Selector并打开ACCEPT

```
SelectionKey key = acceptorSvr.register( selector,  
SelectionKey.OP_ACCEPT, ioHandler);
```

创建并打开run并打开Key

ByteBuffer channel = ByteBuffer.allocate(1024);

```
int readNumber = channel.read(receivedBuffer);
```

```
ByteBuffer byteBuffer = ByteBuffer.allocate(1024);
byteBuffer.reset();
Task task = new Task();
```

```
Object message = null;
while(buffer.hasRemain())
{
    byteBuffer.mark();
    Object message = decode(byteBuffer);
    if (message == null)
    {
        byteBuffer.reset();
        break;
    }
    messageList.add(message );
}
if (!byteBuffer.hasRemain())
byteBuffer.clear();
else
    byteBuffer.compact();
if (messageList != null & !messageList.isEmpty())
{
    for(Object messageE : messageList)
```

```
        handlerTask(messageE);
    }
}
```

POJO encode ByteBuffer SocketChannel write

```
socketChannel.write(buffer);
```

TCP Netty Netty

NIO NIO

2.3.3 NIO TimeServer

TimeServer NIO

2-8 NIO TimeServer

```
9.    public class TimeServer {
10.
11.        /**
12.         * @param args
13.         * @throws IOException
14.         */
15.        public static void main(String[] args) throws
```

```

IOException {
    16.         int port = 8080;
    17.         if (args != null && args.length > 0) {
    18.             try {
    19.                 port = Integer.valueOf(args[0]);
    20.             } catch (NumberFormatException e) {
    21.                 // 打印异常
    22.             }
    23.         }
    24.         MultiplexerTimeServer timeServer = new
MultiplexerTimeServer (port);
        25.         New Thread(timeServer, "NIO-
MultiplexerTimeServer-001").start();
    26.     }
    27. }

```

在NIO中，TimeServer类中，16~23行代码，我们定义了
 24~25行代码，我们创建了一个MultiplexerTimeServer
 对象，并启动了一个Selector线程，这个线程就是
 MultiplexerTimeServer类。

2-8 NIO中的MultiplexerTimeServer

```

    17.     public class MultiplexerTimeServer implements
Runnable {
    18.

```

```

19.         private Selector selector;
20.
21.         private ServerSocketChannel servChannel;
22.
23.         private volatile boolean stop;
24.
25.         /**
26.          * 服务器端
27.          *
28.          * @param port
29.          */
30.         public MultiplexerTimeServer(int port) {
31.             try {
32.                 selector = Selector.open();
33.                 servChannel = ServerSocketChannel.open();
34.                 servChannel.configureBlocking(false);
35.                 servChannel.socket().bind(new
InetSocketAddress(port), 1024);
36.                 servChannel.register(selector,
SelectionKey.OP_ACCEPT);
37.                 System.out.println("The time server is start
in port : " + port);
38.             } catch (IOException e) {
39.                 e.printStackTrace();
40.                 System.exit(1);
41.             }
42.         }

```

```
43.
44.     public void stop() {
45.         this.stop = true;
46.     }
47.
48.     /*
49.      * (non-Javadoc)
50.      *
51.      * @see java.lang.Runnable#run()
52.      */
53.     @Override
54.     public void run() {
55.         while (!stop) {
56.             try {
57.                 selector.select(1000);
58.                 Set<SelectionKey> selectedKeys =
selector.selectedKeys();
59.                 Iterator<SelectionKey> it =
selectedKeys.iterator();
60.                 SelectionKey key = null;
61.                 while (it.hasNext()) {
62.                     key = it.next();
63.                     it.remove();
64.                     try {
65.                         handleInput(key);
66.                     } catch (Exception e) {
67.                         if (key != null) {
```

```

68.            key.cancel();
69.            if (key.channel() != null)
70.                key.channel().close();
71.        }
72.    }
73. }
74. } catch (Throwable t) {
75.     t.printStackTrace();
76. }
77. }
78.
79. // []Channel[]Pipe[]
[]
80.     if (selector != null)
81.         try {
82.             selector.close();
83.         } catch (IOException e) {
84.             e.printStackTrace();
85.         }
86.     }
87.
88.     private void handleInput(SelectionKey key) throws
IOException {
89.
90.         if (key.isValid()) {
91.             // []
92.             if (key.isAcceptable()) {

```



```

93.          // Accept the new connection
          94.          ServerSocketChannel ssc =
(ServerSocketChannel) key.channel();
95.          SocketChannel sc = ssc.accept();
96.          sc.configureBlocking(false);
97.          // Add the new connection to the selector
98.          sc.register(selector, SelectionKey.OP_READ);
99.      }
100.         if (key.isReadable()) {
101.             // Read the data
102.             SocketChannel sc = (SocketChannel)
key.channel();
103.             ByteBuffer readBuffer =
ByteBuffer.allocate(1024);
104.             int readBytes = sc.read(readBuffer);
105.             if (readBytes > 0) {
106.                 readBuffer.flip();
107.                 byte[] bytes = new
byte[readBuffer.remaining()];
108.                 readBuffer.get(bytes);
109.                 String body = new String(bytes, "UTF-
8");
110.                 System.out.println("The time server
receive order : "
111.                     + body);
112.                 String currentTime = "QUERY TIME ORDER"
113.                     .equalsIgnoreCase(body) ? new

```

```
java.util.Date(
```

114.

```
System.currentTimeMillis()).toString()
```

```
115.                : "BAD ORDER";
```

```
116.                doWrite(sc, currentTime);
```

```
117.            } else if (readBytes < 0) {
```

```
118.                // [][][][]
```

```
119.                key.cancel();
```

```
120.                sc.close();
```

```
121.            } else
```

```
122.                ; // []0[][][]
```

```
123.            }
```

```
124.        }
```

```
125.    }
```

```
126.
```

```
127.        private void doWrite(SocketChannel channel,  
String response)
```

```
128.            throws IOException {
```

```
129.                if (response != null &&  
response.trim().length() > 0) {
```

```
130.                    byte[] bytes = response.getBytes();
```

```
131.                    ByteBuffer writeBuffer =  
ByteBuffer.allocate(bytes.length);
```

```
132.                    writeBuffer.put(bytes);
```

```
133.                    writeBuffer.flip();
```

```
134.                    channel.write(writeBuffer);
```

```
135.            }
```

137. }

```

    4100125ByteBuffer
    1MSocketChannel
readSocketChannel
read

```

- 位置0から読み出す
- 位置0から読み出す
- 位置-1から読み出すSocketChannel

readBufferをflipしてlimitをpositionにpositionを0に設定してByteBufferのgetで"QUERY TIME ORDER"を読み出す

5127135をByteBufferにByteBuffer.putでflipしてSocketChannel.writeでSocketChannelで" "をSelectorでByteBuffer.hasRemain()で" "を読み出す

NIO TimeServerでNIOで

2.3.4 NIO

NIO 2-11

2-11 NIO

SocketChannelで

```
SocketChannel clientChannel = SocketChannel.open();
```

```
    // SocketChannel implements TCP connection
    //
```

```
    clientChannel.configureBlocking(false);
    socket.setReuseAddress(true);
    socket.setReceiveBufferSize(BUFFER_SIZE);
    socket.setSendBufferSize(BUFFER_SIZE);
```

```
    // Connect to the server
```

```
        boolean connected=clientChannel.connect(new
InetSocketAddress("ip",port));
```

```
    // Register the channel with the selector
    // Set the key to OP_READ
    // Set the handler to false
    // Set the sync to true
    // Set the ack to true
    //
```

```
    if (connected)
    {
        clientChannel.register( selector, SelectionKey.OP_READ,
ioHandler);
    }
```

```
else
{
    clientChannel.register( selector,
SelectionKey.OP_CONNECT, ioHandler);
}
```

ReactorOP_CONNECTTCP
ACK

```
clientChannel.register( selector, SelectionKey.OP_CONNECT,
ioHandler);
```

Reactor

```
Selector selector = Selector.open();
New Thread(new ReactorTask()).start();
```

runKey

```
int num = selector.select();
Set selectedKeys = selector.selectedKeys();
Iterator it = selectedKeys.iterator();
while (it.hasNext()) {
    SelectionKey key = (SelectionKey)it.next();
```

```
        // ... deal with I/O event ...
    }
```

```
        connect()
```

```
        if (key.isConnectable())
            //handlerConnect();
```

```
        
```

```
        if (channel.finishConnect())
            registerRead();
```

```
        
```

```
        clientChannel.register( selector, SelectionKey.OP_READ,
ioHandler);
```

```
        
```

```
        int  readNumber =  channel.read(receivedBuffer);
```

```
        ByteBuffer resetBuffer = ByteBuffer.allocate(1024);
        Task task = new Task() {
```

```

Object message = null;
while(buffer.hasRemain())
{
    byteBuffer.mark();
    Object message = decode(byteBuffer);
    if (message == null)
    {
        byteBuffer.reset();
        break;
    }
    messageList.add(message );
}
if (!byteBuffer.hasRemain())
byteBuffer.clear();
else
    byteBuffer.compact();
if (messageList != null & !messageList.isEmpty())
{
    for(Object messageE : messageList)
        handlerTask(messageE);
}

```

```

        POJO.encode(ByteBuffer) SocketChannel
write

```

```

socketChannel.write(buffer);

```



```
1.  package com.phei.netty.nio;
2.  import java.io.IOException;
3.  import java.net.InetSocketAddress;
4.  import java.nio.ByteBuffer;
5.  import java.nio.channels.SelectionKey;
6.  import java.nio.channels.Selector;
7.  import java.nio.channels.SocketChannel;
8.  import java.util.Iterator;
9.  import java.util.Set;
10.
11.  /**
12.   * @author Administrator
13.   * @date 2014-2-16
14.   * @version 1.0
15.   */
16.  public class TimeClientHandle implements Runnable {
17.      private String host;
18.      private int port;
19.      private Selector selector;
20.      private SocketChannel socketChannel;
21.      private volatile boolean stop;
22.
23.      public TimeClientHandle(String host, int port) {
24.          this.host = host == null ? "127.0.0.1" : host;
25.          this.port = port;
26.          try {
27.              selector = Selector.open();
```

```

28.         socketChannel = SocketChannel.open();
29.         socketChannel.configureBlocking(false);
30.     } catch (IOException e) {
31.         e.printStackTrace();
32.         System.exit(1);
33.     }
34. }
35.
36. /*
37.  * (non-Javadoc)
38.  *
39.  * @see java.lang.Runnable#run()
40.  */
41. @Override
42. public void run() {
43.     try {
44.         doConnect();
45.     } catch (IOException e) {
46.         e.printStackTrace();
47.         System.exit(1);
48.     }
49.     while (!stop) {
50.         try {
51.             selector.select(1000);
52.             Set<SelectionKey> selectedKeys =
selector.selectedKeys();
53.             Iterator<SelectionKey> it =

```

```

selectedKeys.iterator();
54.         SelectionKey key = null;
55.         while (it.hasNext()) {
56.             key = it.next();
57.             it.remove();
58.             try {
59.                 handleInput(key);
60.             } catch (Exception e) {
61.                 if (key != null) {
62.                     key.cancel();
63.                     if (key.channel() != null)
64.                         key.channel().close();
65.                 }
66.             }
67.         }
68.     } catch (Exception e) {
69.         e.printStackTrace();
70.         System.exit(1);
71.     }
72. }
73.
74. // []Channel[]Pipe[]
[]
75.     if (selector != null)
76.         try {
77.             selector.close();
78.         } catch (IOException e) {

```

```

79.          e.printStackTrace();
80.      }
81.  }
82.
83.      private void handleInput(SelectionKey key) throws
IOException {
84.
85.          if (key.isValid()) {
86.              // 连接成功
87.                  SocketChannel sc = (SocketChannel)
key.channel();
88.                  if (key.isConnectable()) {
89.                      if (sc.finishConnect()) {
90.                          sc.register(selector,
SelectionKey.OP_READ);
91.                          doWrite(sc);
92.                      } else
93.                          System.exit(1); // 连接失败
94.                  }
95.                  if (key.isReadable()) {
96.                      ByteBuffer readBuffer =
ByteBuffer.allocate(1024);
97.                      int readBytes = sc.read(readBuffer);
98.                      if (readBytes > 0) {
99.                          readBuffer.flip();
100.                             byte[] bytes = new
byte[readBuffer.remaining()];

```

```

101.            readBuffer.get(bytes);
102.            String body = new String(bytes, "UTF-
8");
103.            System.out.println("Now is : " + body);
104.            this.stop = true;
105.        } else if (readBytes < 0) {
106.            // 
107.            key.cancel();
108.            sc.close();
109.        } else
110.            ; // 
111.        }
112.    }
113.
114.    }
115.
116.    private void doConnect() throws IOException {
117.        // 
118.            if(socketChannel.connect(new
InetSocketAddress(host, port))) {
119.                socketChannel.register(selector,
SelectionKey.OP_READ);
120.                doWrite(socketChannel);
121.            } else
122.                socketChannel.register(selector,
SelectionKey.OP_CONNECT);
123.        }

```

```

124.
125.         private void doWrite(SocketChannel sc) throws
IOException {
126.             byte[] req = "QUERY TIME ORDER".getBytes();
127.             ByteBuffer writeBuffer =
ByteBuffer.allocate(req.length);
128.             writeBuffer.put(req);
129.             writeBuffer.flip();
130.             sc.write(writeBuffer);
131.             if (!writeBuffer.hasRemaining())
132.                 System.out.println("Send order 2 server
succeed.");
133.         }
134.     }

```

本章主要介绍了 NIO 的 API

1. 2.3.3 介绍了 NIO 的 SocketChannel 接口
 SocketChannel 接口是 2.3.3 中定义的
 SocketChannel 接口是 TCP 接口

2. 4.3.48 介绍了 NIO 的 doConnect 方法
 116 123 介绍了 SocketChannel 的 connect() 方法
 Selector 的 SelectionKey.OP_READ 方法
 TCP 接口 SocketChannel

通过Selector通过SelectionKey.OP_CONNECT来检测TCP
syn-ack通过Selector来检测SocketChannel

34972通过Selector来检测Channel
59通过handleInput(key)来检测handleInput

4通过83通过SelectionKey来检测
ACK通过SocketChannel
finishConnect()truefalse
IOExceptiontrue
SocketChannelSelectionKey.OP_READ
来检测

通过doWrite(sc)125
SocketChannelwrite
“”hasRemaining()
"Send order 2 server succeed."

5通过95
SocketChannel
1MSocketChannelread()
2.3.3
stoptrue

6通过“”7580
Channelpipe
JDKJDKAPI DOC2-12
通过

图2-12 使用Selector的模型

使用Selector的模型NIO模型中，Selector负责从多个通道中读取数据，并将数据放入缓冲区中。

图2-13 使用Selector的模型

图2-13 NIO模型中Selector的作用

图2-14 使用Selector的模型

图2-14 NIO模型中Selector的作用

使用Selector的模型NIO模型中，Selector负责从多个通道中读取数据，并将数据放入缓冲区中。Selector负责从多个通道中读取数据，并将数据放入缓冲区中。

1. 使用Selector的模型NIO模型中，Selector负责从多个通道中读取数据，并将数据放入缓冲区中。

2. SocketChannel使用Selector的模型NIO模型中，Selector负责从多个通道中读取数据，并将数据放入缓冲区中。

3. 使用Selector的模型NIO模型中，Selector负责从多个通道中读取数据，并将数据放入缓冲区中。

JDK1.7中，NIO模型中，Selector负责从多个通道中读取数据，并将数据放入缓冲区中。

看看NIO2.0的AIO

2.4 AIO

NIO2.0的AIO是异步非阻塞I/O，它提供了比NIO更高级的API，使用起来更简单，效率更高。

- `java.util.concurrent.Future`
- `java.nio.channels`

`CompletionHandler`

NIO2.0的AIO提供了比NIO更高级的API，使用起来更简单，效率更高。它提供了比NIO更高级的API，使用起来更简单，效率更高。它提供了比NIO更高级的API，使用起来更简单，效率更高。

看看NIO2.0 AIO

2.4.1 AIO TimeServer

看看

图2-11 AIO TimeClientHandle

```
10. public class TimeServer {
11.
12.     /**
13.      * @param args
14.      * @throws IOException
```

```

15.         */
16.         public static void main(String[] args) throws
IOException {
17.             int port = 8080;
18.             if (args != null && args.length > 0) {
19.                 try {
20.                     port = Integer.valueOf(args[0]);
21.                 } catch (NumberFormatException e) {
22.                     // 打印异常
23.                 }
24.             }
25.             AsyncTimeServerHandler timeServer=new
AsyncTimeServerHandler(port);
26.             new Thread(timeServer, "AIO-
AsyncTimeServerHandler-001").start();
27.         }
28.     }

```

25 打印异常
 AsyncTimeServerHandler

2-12 AIO

```

13.         public class AsyncTimeServerHandler implements
Runnable {
14.

```

```

15.         private int port;
16.
17.         CountDownLatch latch;
18.         AsynchronousServerSocketChannel
asynchronousServerSocketChannel;
19.
20.         public AsyncTimeServerHandler(int port) {
21.             this.port = port;
22.             try {
23.                 asynchronousServerSocketChannel =
AsynchronousServerSocketChannel
24.                     .open();
25.                 asynchronousServerSocketChannel.bind(new
InetSocketAddress(port));
26.                 System.out.println("The time server is start
in port : " + port);
27.             } catch (IOException e) {
28.                 e.printStackTrace();
29.             }
30.         }
31.
32.         /*
33.          * (non-Javadoc)
34.          *
35.          * @see java.lang.Runnable#run()
36.          */
37.         @Override

```


accept() AcceptCompletionHandler
handler() AcceptCompletionHandler

2-13 AIO AcceptCompletionHandler

```
14.  
15.    @Override  
16.    public void completed(AsynchronousSocketChannel  
result,  
17.        AsyncTimeServerHandler attachment) {  
18.  
attachment.asynchronousServerSocketChannel().accept(attachment,  
this);  
19.        ByteBuffer buffer = ByteBuffer.allocate(1024);  
20.        result.read(buffer, buffer, new  
ReadCompletionHandler(result));  
21.    }  
22.  
23.    @Override  
24.    public void failed(Throwable  
exc, AsyncTimeServerHandler attachment) {  
25.        exc.printStackTrace();  
26.        attachment.latch.countDown();  
27.    }  
28. }
```

CompletionHandler

- public void completed(AsynchronousSocketChannel result, AsyncTimeServerHandler attachment)
- public void failed(Throwable exc, AsyncTimeServerHandler attachment)

completed 18 20
attachment AsynchronousServerSocketChannel
accept
accept
AsynchronousServerSocketChannel accept
CompletionHandler completed
AsynchronousServerSocket Channel
accept
read

19
ByteBuffer 1M 20
AsynchronousSocketChannel read read

- ByteBuffer dst Channel
- A attachment Channel
- CompletionHandler Integer, ? super A handler ReadCompletionHandler

ReadCompletionHandler

例2-14 AIO非阻塞式ReadCompletionHandler

```
8.
9.    /**
10.     * @author lilinfeng
11.     * @date 2014-2-16
12.     * @version 1.0
13.     */
14.    public class ReadCompletionHandler implements
15.        CompletionHandler<Integer, ByteBuffer> {
16.
17.        private AsynchronousSocketChannel channel;
18.
19.                                public
ReadCompletionHandler(AsynchronousSocketChannel channel) {
20.            if (this.channel == null)
21.                this.channel = channel;
22.        }
23.
24.        @Override
25.        public void completed(Integer result, ByteBuffer
attachment) {
26.            attachment.flip();
27.            byte[] body = new byte[attachment.remaining()];
28.            attachment.get(body);
29.            try {
```



```

30.            String req = new String(body, "UTF-8");
31.            System.out.println("The time server receive
order : " + req);
32.            String currentTime = "QUERY TIME
ORDER".equalsIgnoreCase(req) ? new java.util.Date(
33.                System.currentTimeMillis()).toString() :
"BAD ORDER";
34.            doWrite(currentTime);
35.        } catch (UnsupportedEncodingException e) {
36.            e.printStackTrace();
37.        }
38.    }
39.
40.    private void doWrite(String currentTime) {
41.        if (currentTime != null &&
currentTime.trim().length() > 0) {
42.            byte[] bytes = (currentTime).getBytes();
43.            ByteBuffer writeBuffer =
ByteBuffer.allocate(bytes.length);
44.            writeBuffer.put(bytes);
45.            writeBuffer.flip();
46.            channel.write(writeBuffer, writeBuffer,
47.                new CompletionHandler<Integer,
ByteBuffer>() {
48.                @Override
49.                public void completed(Integer result,
ByteBuffer buffer) {

```

```

50.                // [][][][][][][][]
51.                if (buffer.hasRemaining())
52.                    channel.write(buffer, buffer, this);
53.            }
54.
55.            @Override
56.                public void failed(Throwable exc,
ByteBuffer attachment) {
57.                try {
58.                    channel.close();
59.                } catch (IOException e) {
60.                    // ignore on close
61.                }
62.            }
63.        });
64.    }
65.}
66.
67.    @Override
68.        public void failed(Throwable exc, ByteBuffer
attachment) {
69.        try {
70.            this.channel.close();
71.        } catch (IOException e) {
72.            e.printStackTrace();
73.        }

```

74. }

75. }

AsynchronousSocketChannel ReadCompletion Handler Netty attachment flip byte new String "QUERY TIME ORDER" doWrite doWrite

41 writeBuffer AsynchronousSocketChannel write read read write CompletionHandler 51 writeBuffer

failed Throwable I/O demo

I/O

2.4.2 AIO TimeClient

例2-15 AIO非同期通信 TimeClient

```
16.         try {
17.             port = Integer.valueOf(args[0]);
18.         } catch (NumberFormatException e) {
19.             // 例外処理
20.         }
21.     }
22.         new Thread(new
AsyncTimeClientHandler("127.0.0.1", port),
23.             "AIO-AsyncTimeClientHandler-001").start();
24.
25.     }
26. }
```

例22は、非同期通信のAIO（Asynchronous I/O）を用いた非同期通信の例を示しています。この例では、JDK 7.0以降のNIO.2（Non-blocking I/O 2.0）のAPI（AsynchronousSocketChannel、AsynchronousDatagramChannel）が利用されています。

例22は、非同期通信のAIO（Asynchronous I/O）を用いた非同期通信の例を示しています。

例2-16 AIO非同期通信 AsyncTimeClientHandler

```
1. package com.phei.netty.aio;
2.
3. import java.io.IOException;
4. import java.io.UnsupportedEncodingException;
```

```

5.  import java.net.InetSocketAddress;
6.  import java.nio.ByteBuffer;
7.  import java.nio.channels.AsynchronousSocketChannel;
8.  import java.nio.channels.CompletionHandler;
9.  import java.util.concurrent.CountDownLatch;
10.
11.  /**
12.   * @author Administrator
13.   * @date 2014-2-16
14.   * @version 1.0
15.   */
16.  public class AsyncTimeClientHandler implements
17.      CompletionHandler<Void, AsyncTimeClientHandler>,
Runnable {
18.
19.      private AsynchronousSocketChannel client;
20.      private String host;
21.      private int port;
22.      private CountDownLatch latch;
23.
24.      public AsyncTimeClientHandler(String host, int
port) {
25.          this.host = host;
26.          this.port = port;
27.          try {
28.              client = AsynchronousSocketChannel.open();
29.          } catch (IOException e) {

```

```

30.         e.printStackTrace();
31.     }
32. }
33.
34.     @Override
35.     public void run() {
36.         latch = new CountDownLatch(1);
37.         client.connect(new InetSocketAddress(host, port),
this, this);
38.         try {
39.             latch.await();
40.         } catch (InterruptedException e1) {
41.             e1.printStackTrace();
42.         }
43.         try {
44.             client.close();
45.         } catch (IOException e) {
46.             e.printStackTrace();
47.         }
48.     }
49.
50.     @Override
51.         public void completed(Void result,
AsyncTimeClientHandler attachment) {
52.         byte[] req = "QUERY TIME ORDER".getBytes();
53.             ByteBuffer writeBuffer =
ByteBuffer.allocate(req.length);

```

```

54.     writeBuffer.put(req);
55.     writeBuffer.flip();
56.     client.write(writeBuffer, writeBuffer,
57.         new CompletionHandler<Integer, ByteBuffer>() {
58.             @Override
59.             public void completed(Integer result,
ByteBuffer buffer) {
60.                 if (buffer.hasRemaining()) {
61.                     client.write(buffer, buffer, this);
62.                 } else {
63.                     ByteBuffer readBuffer =
ByteBuffer.allocate(1024);
64.                     client.read(
65.                         readBuffer,
66.                         readBuffer,
67.                         new CompletionHandler<Integer,
ByteBuffer>() {
68.                             @Override
69.                             public void completed(Integer
result,
70.                                 ByteBuffer buffer) {
71.                                     buffer.flip();
72.                                     byte[] bytes = new byte[buffer
73.                                         .remaining()];
74.                                     buffer.get(bytes);
75.                                     String body;
76.                                     try {

```

```

77.                body = new String(bytes,
78.                    "UTF-8");
79.                System.out.println("Now is : "
80.                    + body);
81.                latch.countDown();
82.                } catch
(UnsupportedEncodingException e) {
83.                    e.printStackTrace();
84.                }
85.            }
86.
87.            @Override
88.            public void failed(Throwable exc,
89.                ByteBuffer attachment) {
90.                try {
91.                    client.close();
92.                    latch.countDown();
93.                } catch (IOException e) {
94.                    // ignore on close
95.                }
96.            }
97.        });
98.    }
99.}
100.
101.        @Override
102.        public void failed(Throwable

```



```

exc,ByteBuffer attachment) {
    103.            try {
    104.                client.close();
    105.                latch.countDown();
    106.            } catch (IOException e) {
    107.                // ignore on close
    108.            }
    109.        }
    110.    });
    111. }
    112.
    113.    @Override
    114.        public void failed(Throwable exc,
AsyncTimeClientHandler attachment) {
    115.        exc.printStackTrace();
    116.        try {
    117.            client.close();
    118.            latch.countDown();
    119.        } catch (IOException e) {
    120.            e.printStackTrace();
    121.        }
    122.    }
    123. }

```

AsyncTimeClientHandler

```

24    32    AsynchronousSocketChannel    open
    AsynchronousSocketChannel    36    CountDownLatch
37    connect

```

- A `Attachment` is an `AsynchronousSocketChannel` that is used to send an attachment to a client.
- A `CompletionHandler` is a `Void, ? super A` handler that is used to handle the completion of an attachment.

```

    AsyncTimeClientHandler
CompletionHandler

```

```

        completed()
        writeBuffer()
        Asynchronous
        SocketChannel
        write
        CompletionHandler
        Integer, ByteBuffer
        60, 62

```

```

    64 97
AsynchronousSocketChannel.read
CompletionHandler Integer
ByteBuffer JDK 71 78
CompletionHandler ByteBuffer

```

```

    102 111  CountDownLatch
countDown AsyncTimeClientHandler

```

在本书第2章中，我们学习了NIO的基本概念，并了解了NIO的三大核心类：Selector、Channel和Buffer。本章我们将进一步探讨NIO的高级应用，包括Netty框架和AIO（Asynchronous I/O）。

Netty是一个高性能的NIO框架，它提供了丰富的API，使得开发网络应用程序变得更加简单。本章我们将介绍Netty的基本用法，并展示如何利用Netty实现一个高性能的NIO应用。

AIO（Asynchronous I/O）是Java 7引入的一个新特性，它允许我们在非阻塞的情况下完成I/O操作。本章我们将介绍AIO的基本用法，并展示如何利用AIO实现一个高性能的NIO应用。

2.4.3 AIO 异步I/O

在TimeServer中，我们使用了2-15的示例代码。

2-15 AIO 异步I/O 示例代码

在TimeClient中，我们使用了2-16的示例代码。

2-16 AIO 异步I/O 示例代码

在TimeServer中，我们使用了JDK 7引入的CompletionHandler接口。

2-17 AIO 异步I/O 示例代码

“Thread-2”线程中，我们使用了JDK 7引入的ThreadPoolExecutor。在sun.nio.ch.AsynchronousChannelGroupImpl中，我们使用了com.phei.netty.aio.AsyncTimeClientHandler\$1.completed。在Socket Channel中，我们使用了NIO。在I/O中，我们使用了AsynchronousServerSocket Channel。

AsynchronousSocketChannel。JDK。NIO2.0ChannelNIO。

JDK1.7I/OAIOJavaJDK1.6NIO2.0NIO2.0JDK1.7JDK1.7。

5I/OI/O。

2.5 4I/O

2.5.1

1I/O

JDK1.4NIOI/OUNIXJDKI/OI/OJDK1.41.5 update10JDKSelectorselect/pollI/OI/OI/OJDK1.5 update10Linux core2.6SunSelctorepollselect/pollAPIJDK NIOI/O2-17。

JDK1.7NIO2.0I/OI/OAIO。

```
NIO[]
NIO[]I/O[]NIO[]
NIO[]I/O[]I/O[]
```

□2-17 JDK1.5_update10□□epoll

2. Selector

```

    Selector
    Selector

```

`ByteBuffer`Java NIO
SelectorChannelChannelChannelChannel
ChannelI/ONIOI/O
SelectorSelector
`ByteBuffer`

3□□□□I/O

```

    public void run() {
        while (true) {
            try {
                // 1. 从队列中取出任务
                Task task = queue.poll();
                if (task == null) {
                    // 队列为空，等待一段时间
                    Thread.sleep(100);
                    continue;
                }
                // 2. 执行任务
                task.execute();
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
}

```

□□□□□□□□□□ I/O □□□□□□□□□□ I/O □□□□

2.5.2 网络I/O模型

网络I/O模型是指API提供的接口和底层实现的组合。图2-1展示了网络I/O API提供的接口和底层实现的组合。

图2-1 网络I/O模型

网络I/O模型是指API提供的接口和底层实现的组合。图2-1展示了网络I/O API提供的接口和底层实现的组合。图2-1展示了网络I/O API提供的接口和底层实现的组合。图2-1展示了网络I/O API提供的接口和底层实现的组合。

图2-1展示了网络I/O API提供的接口和底层实现的组合。图2-1展示了网络I/O API提供的接口和底层实现的组合。图2-1展示了网络I/O API提供的接口和底层实现的组合。

2.6 网络Netty模型

网络I/O模型是指API提供的接口和底层实现的组合。图2-1展示了网络I/O API提供的接口和底层实现的组合。图2-1展示了网络I/O API提供的接口和底层实现的组合。图2-1展示了网络I/O API提供的接口和底层实现的组合。

网络I/O模型是指API提供的接口和底层实现的组合。图2-1展示了网络I/O API提供的接口和底层实现的组合。图2-1展示了网络I/O API提供的接口和底层实现的组合。图2-1展示了网络I/O API提供的接口和底层实现的组合。

本章主要介绍NIO的I/O模型，以及NIO的API。本章主要介绍NIO的I/O模型，以及NIO的API。

2.6.1 Java NIO

本章主要介绍JDK NIO的API。

1 NIO API 主要包含 Selector, ServerSocketChannel, SocketChannel, ByteBuffer。

2 本章主要介绍Java NIO的Reactor模型。

3 本章主要介绍NIO的API。

4 JDK NIO BUG 主要包含 epoll bug, Selector, CPU 100%, JDK1.6 update18, JDK1.7, BUG, BUG, BUG。

- http://bugs.java.com/bugdatabase/view_bug.do?bug_id=6403933
- http://bugs.java.com/bugdatabase/view_bug.do?bug_id=2147719

本章主要介绍NIO的API。

java.lang.Thread.State: RUNNABLE

at sun.nio.ch.EPollArrayWrapper.epollWait(Native Method)

at

sun.nio.ch.EPollArrayWrapper.poll(EPollArrayWrapper.java:210)

at

***sun.nio.ch.EPollSelectorImpl.doSelect(EPollSelectorImpl.java:65
)***

at

sun.nio.ch.SelectorImpl.lockAndDoSelect(SelectorImpl.java:69)

- locked <0x0000000750928190> (a sun.nio.ch.Util\$2)

Netty

2.6.2 Netty

Netty NIO Hadoop RPC avro Netty RPC Netty

Netty

- API
- ChannelHandler
- NIO Netty
- Netty JDK NIO BUG
- BUG
- Netty

Netty Java NIO

2.7

demo I/O I/O I/O NIO I/O AIO

JDK NIO I/O Netty Netty NIO

Netty 是 Java 中实现 NIO 的框架，Netty 是 NIO 的框架。

Netty NIO

3 Netty

4 TCP/

5

第3章 Netty入门

本章Netty入门分为两部分，第一部分介绍Netty入门，第二部分介绍Netty入门。

第一部分介绍Netty入门。

第二部分介绍

- Netty入门
- 时间服务器TimeServer
- 时间客户端TimeClient
- 时间客户端

3.1 Netty入门

本章介绍Netty入门，包括JDK1.7和JDK8的path，以及IDE Eclipse和Java IDE NetBeans。

本章介绍Netty入门，包括JDK1.7和JDK8的path，以及IDE Eclipse和Java IDE NetBeans。

本章介绍Netty入门。

3.1.1 Netty入门

Netty 请到 <http://netty.io/> 的 Downloads 页面下载 5.0.0.Alpha1 版本，大小为 8.95M，如图 3-1 所示。

图 3-1 Netty 5.0 下载

将下载的 netty-all-5.0.0.Alpha1.jar 文件放入 lib 目录，如图 3-2 所示。

图 3-2 Netty 5.0 文件放入 lib 目录

将 netty-all-5.0.0.Alpha1.jar 文件放入 lib 目录，如图 3-2 所示。

3.1.2 配置 Netty

在 Eclipse 中配置 Java 项目，如图 3-3 所示。

图 3-3 Netty 配置

将 netty-all-5.0.0.Alpha1.jar 文件放入 lib 目录，如图 3-4 所示。

图 3-4 添加 netty jar 文件

将 netty-all-5.0.0.Alpha1.jar 文件放入 lib 目录，如图 3-4 所示。

图 3-5 配置 Netty.jar 的 ClassPath

将 Netty 项目配置为 Maven 项目，如图 3-5 所示。

3.2 Netty

NettyNetty2

TimeServer

NettyTimeServerNIO

1 ServerSocketChannel

2 TCP backlog

3 I/O Selector

```

    4 Selector ServerSocketChannel Selector
    SelectionKey.ACCEPT

```

```
5 // I/O Selector.select() Channel
```

```

    6Channel.OP_ACCEPT
    ServerSocketChannel.accept()

```

```
07SocketChannelTCP
```

```
08 SocketChannel Selector OP_READ
```

```

    9ChannelOP_READSocketChannel
ByteBuffer

```

10 Channel.OP_WRITE

NIO JDK NIO
Netty NIO
Netty

3-1 Netty TimeServer

```
16. public class TimeServer {
17.
18.     public void bind(int port) throws Exception {
19.         // NIO
20.         EventLoopGroup bossGroup = new
NioEventLoopGroup();
21.         EventLoopGroup workerGroup = new
NioEventLoopGroup();
22.         try {
23.             ServerBootstrap b = new ServerBootstrap();
24.             b.group(bossGroup, workerGroup)
25.                 .channel(NioServerSocketChannel.class)
26.                 .option(ChannelOption.SO_BACKLOG, 1024)
27.                 .childHandler(new ChildChannelHandler());
28.             //
29.             ChannelFuture f = b.bind(port).sync();
30.
31.             //
32.             f.channel().closeFuture().sync();
```

```

33.         } finally {
34.             // 關閉服務器
35.             bossGroup.shutdownGracefully();
36.             workerGroup.shutdownGracefully();
37.         }
38.     }
39.
40.         private class ChildChannelHandler extends
ChannelInitializer <SocketChannel> {
41.             @Override
42.             protected void initChannel(SocketChannel arg0)
throws Exception {
43.                 arg0.pipeline().addLast(new
TimeServerHandler());
44.             }
45.
46.         }
47.
48.         /**
49.          * @param args
50.          * @throws Exception
51.          */
52.         public static void main(String[] args) throws
Exception {
53.             int port = 8080;
54.             if (args != null && args.length > 0) {
55.                 try {

```

```

56.         port = Integer.valueOf(args[0]);
57.     } catch (NumberFormatException e) {
58.         // 忽略
59.     }
60. }
61. new TimeServer().bind(port);
62. }
63. }

```

在 Netty 中，我们使用 `Netty` 来创建 `Channel`。
 在 `Netty` 中，我们使用 `Channel` 来创建 `Channel`。

在 `bind` 方法中，我们使用 `20` 和 `21` 来创建 `NioEventLoopGroup`。
 在 `NioEventLoopGroup` 中，我们使用 `NIO` 来创建 `Reactor`。
 在 `Reactor` 中，我们使用 `SocketChannel` 来创建 `23`。
 在 `ServerBootstrap` 中，我们使用 `Netty`。
 在 `NIO` 中，我们使用 `24`。
 在 `ServerBootstrap` 中，我们使用 `group`。
 在 `NIO` 中，我们使用 `ServerBootstrap`。
 在 `Channel` 中，我们使用 `NioServerSocketChannel`。
 在 `JDK NIO` 中，我们使用 `ServerSocketChannel`。
 在 `NioServerSocketChannel` 中，我们使用 `TCP`。
 在 `backlog` 中，我们使用 `1024`。
 在 `I/O` 中，我们使用 `ChildChannelHandler`。
 在 `Reactor` 中，我们使用 `handler`。
 在 `I/O` 中，我们使用 `ChannelFuture`。

在 `bind` 方法中，我们使用 `sync` 来创建 `Netty`。
 在 `ChannelFuture` 中，我们使用 `JDK`。

java.util.concurrent.Future

32 f.channel().closeFuture().sync()
main

34 36 NIO shutdownGracefully
shutdownGracefully

TimeServerHandler

3-2 Netty TimeServerHandler

```
12.      public class TimeServerHandler extends
ChannelHandlerAdapter {
13.
14.      @Override
15.      public void channelRead(ChannelHandlerContext
ctx, Object msg)
16.      throws Exception {
17.      ByteBuf buf = (ByteBuf) msg;
18.      byte[] req = new byte[buf.readableBytes()];
19.      buf.readBytes(req);
20.      String body = new String(req, "UTF-8");
21.      System.out.println("The time server receive order
: " + body);
22.      String currentTime = "QUERY TIME
ORDER".equalsIgnoreCase(body) ? new java.util.Date(
23.      System.currentTimeMillis()).toString() : "BAD
```

```

ORDER";

        24.                ByteBuf    resp    =
Unpooled.copiedBuffer(currentTime.getBytes());
        25.                ctx.write(resp);
        26.            }
        27.
        28.            @Override
                29.                public void
channelReadComplete(ChannelHandlerContext ctx) throws Exception
{
        30.                ctx.flush();
        31.            }
        32.
        33.            @Override
        34.                public void exceptionCaught(ChannelHandlerContext
ctx, Throwable cause) {
        35.                ctx.close();
        36.            }
        37.    }

```

TimeServerHandler 实现 ChannelHandlerAdapter 接口
 实现 channelRead 和 exceptionCaught 方法

17 行代码，msg 是 Netty 的 ByteBuf，ByteBuf 是 JDK 的
 java.nio.ByteBuffer，ByteBuf 的 readableBytes 方法返回 byte 数组

```
ByteBuf readBytes(ByteBuf byte) new
String("QUERY TIME ORDER")
ChannelHandlerContext write
```

```
30 ChannelHandlerContext flush
SocketChannel
Selector Netty write SocketChannel
write flush
SocketChannel
```

```
35 ChannelHandlerContext
ChannelHandlerContext
```

```
30 NIO
JDK NIO
```

```
Netty TimeClient
```

3.3 Netty

```
Netty
```

TimeClient

3-3 Netty TimeClient

```
16. public class TimeClient {
17.
18.     public void connect(int port, String host) throws
```



```

Exception {
    19.         // 创建NIO线程
    20.         EventLoopGroup group = new NioEventLoopGroup();
    21.         try {
    22.             Bootstrap b = new Bootstrap();
    23.
    b.group(group).channel(NioSocketChannel.class)
    24.                 .option(ChannelOption.TCP_NODELAY, true)
    25.                 .handler(new
ChannelInitializer<SocketChannel>() {
    26.                     @Override
    27.                     public void initChannel(SocketChannel ch)
    28.                         throws Exception {
    29.                         ch.pipeline().addLast(new
TimeClientHandler());
    30.                     }
    31.                 });
    32.
    33.         // 创建服务器
    34.             ChannelFuture f = b.connect(host,
port).sync();
    35.
    36.         // 关闭服务器
    37.         f.channel().closeFuture().sync();
    38.     } finally {
    39.         // 关闭NIO线程
    40.         group.shutdownGracefully();

```

```

41.     }
42.     }
43.
44.     /**
45.      * @param args
46.      * @throws Exception
47.      */
48.     public static void main(String[] args) throws
Exception {
49.         int port = 8080;
50.         if (args != null && args.length > 0) {
51.             try {
52.                 port = Integer.valueOf(args[0]);
53.             } catch (NumberFormatException e) {
54.                 // 打印错误
55.             }
56.         }
57.         new TimeClient().connect(port, "127.0.0.1");
58.     }
59. }

```

调用connect方法时，20秒内如果没有建立连接，则抛出TimeoutException异常。
 在main方法中，我们使用NioEventLoopGroup来创建NioSocketChannel，并为其指定handler。
 在initChannel方法中，我们使用NioSocketChannel来创建NioSocketChannel。

ChannelHandlerChannelPipelineI/O

connect

NIO

TimeClientHandler

3-4 NettyTimeClientHandler

```
14.      public class TimeClientHandler extends
ChannelHandlerAdapter {
15.
16.      private static final Logger logger = Logger
17.
.getLogger(TimeClientHandler.class.getName());
18.
19.      private final ByteBuf firstMessage;
20.
21.      /**
22.       * Creates a client-side handler.
23.       */
24.      public TimeClientHandler() {
25.          byte[] req = "QUERY TIME ORDER".getBytes();
26.          firstMessage = Unpooled.buffer(req.length);
27.          firstMessage.writeBytes(req);
```

```

28.
29.     }
30.
31.     @Override
32.     public void channelActive(ChannelHandlerContext
ctx) {
33.         ctx.writeAndFlush(firstMessage);
34.     }
35.
36.     @Override
37.     public void channelRead(ChannelHandlerContext
ctx, Object msg)
38.         throws Exception {
39.         ByteBuf buf = (ByteBuf) msg;
40.         byte[] req = new byte[buf.readableBytes()];
41.         buf.readBytes(req);
42.         String body = new String(req, "UTF-8");
43.         System.out.println("Now is : " + body);
44.     }
45.
46.     @Override
47.     public void exceptionCaught(ChannelHandlerContext
ctx, Throwable cause) {
48.         // 打印异常
49.         logger.warning("Unexpected exception from
downstream : "
50.             + cause.getMessage());

```

```

51.         ctx.close();
52.     }
53. }

```

在 `channelActive` 和 `channelRead` 方法中，`exceptionCaught` 方法用于处理 TCP 连接异常。Netty 的 `NIO` 模型中，`channelActive` 方法用于处理 `ChannelHandlerContext` 的 `writeAndFlush` 方法。

在 `channelRead` 方法中，`39` 和 `43` 行分别调用了 `Netty` 的 `ByteBuf` 方法。

`47` 和 `52` 行分别调用了 `Netty` 的 `ByteBuf` 方法。

3.4 本章小结

3.4.1 本章小结

在 Eclipse 中，我们使用 `Java` 运行 `TimeServer`。在 `TimeServer` 类中，我们使用 `Run` 方法。在 `As→Java Application` 菜单中，我们使用 `Alt+Shift+X` 快捷键。

图 3-6 TimeServer

在 `TimeServer` 类中，我们使用 `Run` 方法。

在 `TimeServer` 类中，我们使用 `Run` 方法。

图 3-7 TimeServer

□□□□□□□□3-8□□□

3-8 TimeClient

Netty NIO API

[illegible]

3.4.2 五五五五五

```

    NettyWebJava.jar
    NettyWebJava.jar

```

```

1 Eclipse\Java.jar

```

```
2 ant Netty.jar
xxx_install.gz
```

3 Maven
11/11/2023

3.5 ☐ ☐

Netty
NIO

111

```

NettyNetty
Netty

```

```
04Netty
Netty
Netty
```


第4章 TCP/IP/网络层

本章TCP/IP网络层主要介绍IP地址、子网划分、路由选择、网络层设备、Netty网络层应用。

本章主要介绍TCP/IP网络层主要概念、Netty网络层应用。

本章主要介绍

- TCP/IP/网络层
- 子网TCP/IP/网络层
- Netty网络层应用

4.1 TCP/IP/网络层

TCP/IP“四”层模型主要介绍IP地址、子网划分、路由选择、网络层设备、Netty网络层应用。

4.1.1 TCP/IP/网络层

本章主要介绍TCP/IP网络层主要概念、4-1


```

14    private int counter;
15
16    @Override
17    public void channelRead(ChannelHandlerContext ctx,
Object msg)
18        throws Exception {
19        ByteBuf buf = (ByteBuf) msg;
20        byte[] req = new byte[buf.readableBytes()];
21        buf.readBytes(req);
22        String body = new String(req, "UTF-8").substring(0,
req.length
23        );
System.getProperty("line.separator").length());
24        System.out.println("The time server receive order :
" + body
25        + " ; the counter is : " + ++counter);
26        String currentTime = "QUERY TIME
ORDER".equalsIgnoreCase(body) ? new java.util.Date(
27        System.currentTimeMillis()).toString() : "BAD
ORDER";
28        currentTime = currentTime +
System.getProperty("line.separator");
29        ByteBuf resp =
Unpooled.copiedBuffer(currentTime.getBytes());
30        ctx.writeAndFlush(resp);
31    }
32

```



```

25    */
26    public TimeClientHandler() {
27        req = ("QUERY TIME ORDER" +
System.getProperty("line.separator"))
28        .getBytes();
29    }
30
31    @Override
32    public void channelActive(ChannelHandlerContext ctx)
{
33        ByteBuf message = null;

34        for (int i = 0; i < 100; i++) {

35        message = Unpooled.buffer(req.length);

```

```
36    message.writeBytes(req);
```

```
37    ctx.writeAndFlush(message);
```

```
38}
```

```
39    }
```

```
40
```

```
41    @Override
```

```
42    public void channelRead(ChannelHandlerContext ctx,  
Object msg)
```

```
43    throws Exception {
```

```
44    ByteBuf buf = (ByteBuf) msg;
```

```
45    byte[] req = new byte[buf.readableBytes()];
```

```
46 buf.readBytes(req);
47 String body = new String(req, "UTF-8");
48 System.out.println("Now is : " + body + " ; the counter
is : "
```

```
49     + ++counter);
```

```
50     }
51
52     @Override
53     public void exceptionCaught(ChannelHandlerContext
ctx, Throwable cause) {
54         // 处理异常
55         logger.warning("Unexpected exception from downstream : "
56             + cause.getMessage());
57         ctx.close();
58     }
59 }
```

□□□□□□□□33□38□□□□□□□□□□□□□□□□100□□□□□□
□□□□□□□□□□□□□□□□Channel□□□□□□□□□□□□□□□□100□□□
□□□□□□□□□

□48□49□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
□□□100□□□□□□□□□□

□□□□□□□□□□□□□□□□□□□□□□

4.2.3 □□□□

□□□□□□□□□□□□□□□□□□□□

□□□□□□□□□□

The time server receive order : QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORD ; the counter is : 1

The time server receive order :

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER

QUERY TIME ORDER ; the counter is : 2

□ □ □ □ □ □ □ □ □ □

```
; the counter is : 1
```

```

    Netty LineBasedFrameDecoder
StringDecoderTCP

```

4.3 `LineBasedFrameDecoder` TCP

`LineBasedFrameDecoder` TCP / Netty TCP NIO API

4.3.1 TCP `TimeServer`

`LineBasedFrameDecoder` `StringDecoder` API

4-3 Netty `TimeServer`

```
18 public class TimeServer {
19
20     public void bind(int port) throws Exception {
21         // NIO
22         EventLoopGroup bossGroup = new NioEventLoopGroup();
23         EventLoopGroup workerGroup = new NioEventLoopGroup();
24         try {
25             ServerBootstrap b = new ServerBootstrap();
26             b.group(bossGroup, workerGroup)
27                 .channel(NioServerSocketChannel.class)
28                 .option(ChannelOption.SO_BACKLOG, 1024)
```

```

29    .childHandler(new ChildChannelHandler());
30    // 關閉通道
31    ChannelFuture f = b.bind(port).sync();
32
33    // 關閉通道
34    f.channel().closeFuture().sync();
35} finally {
36    // 關閉通道
37    bossGroup.shutdownGracefully();
38    workerGroup.shutdownGracefully();
39}
40    }
41
42    private class ChildChannelHandler extends
ChannelInitializer <SocketChannel> {
43    @Override
44    protected void initChannel(SocketChannel arg0) throws
Exception {
45        arg0.pipeline().addLast(new
LineBasedFrameDecoder(1024));

46        arg0.pipeline().addLast(new StringDecoder());

```

```
47    arg0.pipeline().addLast(new TimeServerHandler());
```

```
48}
```

```
49    }
```

```
50
```

```
51    /**
```

```
52        * @param args
```

```
53        * @throws Exception
```

```
54        */
```

```
55    public static void main(String[] args) throws  
Exception {
```

```
56    int port = 8080;
```

```
57    if (args != null && args.length > 0) {
```

```
58        try {
```

```
59            port = Integer.valueOf(args[0]);
```

```
60        } catch (NumberFormatException e) {
```

```
61            // 打印异常
```

```
62        }
```

```
63    }
```



```

64    new TimeServer().bind(port);
65    }
66}

```

45 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207 208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223 224 225 226 227 228 229 230 231 232 233 234 235 236 237 238 239 240 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255 256 257 258 259 260 261 262 263 264 265 266 267 268 269 270 271 272 273 274 275 276 277 278 279 280 281 282 283 284 285 286 287 288 289 290 291 292 293 294 295 296 297 298 299 300 301 302 303 304 305 306 307 308 309 310 311 312 313 314 315 316 317 318 319 320 321 322 323 324 325 326 327 328 329 330 331 332 333 334 335 336 337 338 339 340 341 342 343 344 345 346 347 348 349 350 351 352 353 354 355 356 357 358 359 360 361 362 363 364 365 366 367 368 369 370 371 372 373 374 375 376 377 378 379 380 381 382 383 384 385 386 387 388 389 390 391 392 393 394 395 396 397 398 399 400 401 402 403 404 405 406 407 408 409 410 411 412 413 414 415 416 417 418 419 420 421 422 423 424 425 426 427 428 429 430 431 432 433 434 435 436 437 438 439 440 441 442 443 444 445 446 447 448 449 450 451 452 453 454 455 456 457 458 459 460 461 462 463 464 465 466 467 468 469 470 471 472 473 474 475 476 477 478 479 480 481 482 483 484 485 486 487 488 489 490 491 492 493 494 495 496 497 498 499 500 501 502 503 504 505 506 507 508 509 510 511 512 513 514 515 516 517 518 519 520 521 522 523 524 525 526 527 528 529 530 531 532 533 534 535 536 537 538 539 540 541 542 543 544 545 546 547 548 549 550 551 552 553 554 555 556 557 558 559 560 561 562 563 564 565 566 567 568 569 570 571 572 573 574 575 576 577 578 579 580 581 582 583 584 585 586 587 588 589 590 591 592 593 594 595 596 597 598 599 600 601 602 603 604 605 606 607 608 609 610 611 612 613 614 615 616 617 618 619 620 621 622 623 624 625 626 627 628 629 630 631 632 633 634 635 636 637 638 639 640 641 642 643 644 645 646 647 648 649 650 651 652 653 654 655 656 657 658 659 660 661 662 663 664 665 666 667 668 669 670 671 672 673 674 675 676 677 678 679 680 681 682 683 684 685 686 687 688 689 690 691 692 693 694 695 696 697 698 699 700 701 702 703 704 705 706 707 708 709 710 711 712 713 714 715 716 717 718 719 720 721 722 723 724 725 726 727 728 729 730 731 732 733 734 735 736 737 738 739 740 741 742 743 744 745 746 747 748 749 750 751 752 753 754 755 756 757 758 759 760 761 762 763 764 765 766 767 768 769 770 771 772 773 774 775 776 777 778 779 780 781 782 783 784 785 786 787 788 789 790 791 792 793 794 795 796 797 798 799 800 801 802 803 804 805 806 807 808 809 810 811 812 813 814 815 816 817 818 819 820 821 822 823 824 825 826 827 828 829 830 831 832 833 834 835 836 837 838 839 840 841 842 843 844 845 846 847 848 849 850 851 852 853 854 855 856 857 858 859 860 861 862 863 864 865 866 867 868 869 870 871 872 873 874 875 876 877 878 879 880 881 882 883 884 885 886 887 888 889 890 891 892 893 894 895 896 897 898 899 900 901 902 903 904 905 906 907 908 909 910 911 912 913 914 915 916 917 918 919 920 921 922 923 924 925 926 927 928 929 930 931 932 933 934 935 936 937 938 939 940 941 942 943 944 945 946 947 948 949 950 951 952 953 954 955 956 957 958 959 960 961 962 963 964 965 966 967 968 969 970 971 972 973 974 975 976 977 978 979 980 981 982 983 984 985 986 987 988 989 990 991 992 993 994 995 996 997 998 999 1000

4-4 Netty 4.4.4 TimeServerHandler

```

12    public class TimeServerHandler extends
ChannelHandlerAdapter {
13
14        private int counter;
15
16        @Override
17        public void channelRead(ChannelHandlerContext ctx,
Object msg)
18            throws Exception {
19            String body = (String) msg;

```

```

20            System.out.println("The time server receive order : " +

```

body

```
21    + " ; the counter is : " + ++counter);
```

```
22    String    currentTime    =    "QUERY    TIME
ORDER".equalsIgnoreCase(body) ? new java.util.Date(
23        System.currentTimeMillis()).toString() : "BAD
ORDER";
```

```
24    currentTime    =    currentTime    +
System.getProperty("line.separator");
```

```
25    ByteBuf    resp    =
Unpooled.copiedBuffer(currentTime.getBytes());
```

```
26    ctx.writeAndFlush(resp);
```

```
27    }
```

```
28
```

```
29    @Override
```

```
30    public void exceptionCaught(ChannelHandlerContext
ctx, Throwable cause) {
```

```
31    ctx.close();
```



```
30      throws Exception {  
31      ch.pipeline().addLast(
```

```
32      new LineBasedFrameDecoder(1024));
```

```
33      ch.pipeline().addLast(new StringDecoder());
```

```
34      ch.pipeline().addLast(new  
TimeClientHandler());
```

```
35      }
```

```

36        });
37
38        // 關閉通道
39        ChannelFuture f = b.connect(host, port).sync();
40
41        // 關閉通道
42        f.channel().closeFuture().sync();
43    } finally {
44        // 關閉NIO通道
45        group.shutdownGracefully();
46    }
47    }
48
49    /**
50     * @param args
51     * @throws Exception
52     */
53    public static void main(String[] args) throws
Exception {
54        int port = 8080;
55        if (args != null && args.length > 0) {
56            try {
57                port = Integer.valueOf(args[0]);
58            } catch (NumberFormatException e) {
59                // 關閉通道
60            }
61        }

```

```

62    new TimeClient().connect(port, "127.0.0.1");
63    }
64}

```

31 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207 208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223 224 225 226 227 228 229 230 231 232 233 234 235 236 237 238 239 240 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255 256 257 258 259 260 261 262 263 264 265 266 267 268 269 270 271 272 273 274 275 276 277 278 279 280 281 282 283 284 285 286 287 288 289 290 291 292 293 294 295 296 297 298 299 300 301 302 303 304 305 306 307 308 309 310 311 312 313 314 315 316 317 318 319 320 321 322 323 324 325 326 327 328 329 330 331 332 333 334 335 336 337 338 339 340 341 342 343 344 345 346 347 348 349 350 351 352 353 354 355 356 357 358 359 360 361 362 363 364 365 366 367 368 369 370 371 372 373 374 375 376 377 378 379 380 381 382 383 384 385 386 387 388 389 390 391 392 393 394 395 396 397 398 399 400 401 402 403 404 405 406 407 408 409 410 411 412 413 414 415 416 417 418 419 420 421 422 423 424 425 426 427 428 429 430 431 432 433 434 435 436 437 438 439 440 441 442 443 444 445 446 447 448 449 450 451 452 453 454 455 456 457 458 459 460 461 462 463 464 465 466 467 468 469 470 471 472 473 474 475 476 477 478 479 480 481 482 483 484 485 486 487 488 489 490 491 492 493 494 495 496 497 498 499 500 501 502 503 504 505 506 507 508 509 510 511 512 513 514 515 516 517 518 519 520 521 522 523 524 525 526 527 528 529 530 531 532 533 534 535 536 537 538 539 540 541 542 543 544 545 546 547 548 549 550 551 552 553 554 555 556 557 558 559 560 561 562 563 564 565 566 567 568 569 570 571 572 573 574 575 576 577 578 579 580 581 582 583 584 585 586 587 588 589 590 591 592 593 594 595 596 597 598 599 600 601 602 603 604 605 606 607 608 609 610 611 612 613 614 615 616 617 618 619 620 621 622 623 624 625 626 627 628 629 630 631 632 633 634 635 636 637 638 639 640 641 642 643 644 645 646 647 648 649 650 651 652 653 654 655 656 657 658 659 660 661 662 663 664 665 666 667 668 669 670 671 672 673 674 675 676 677 678 679 680 681 682 683 684 685 686 687 688 689 690 691 692 693 694 695 696 697 698 699 700 701 702 703 704 705 706 707 708 709 710 711 712 713 714 715 716 717 718 719 720 721 722 723 724 725 726 727 728 729 730 731 732 733 734 735 736 737 738 739 740 741 742 743 744 745 746 747 748 749 750 751 752 753 754 755 756 757 758 759 760 761 762 763 764 765 766 767 768 769 770 771 772 773 774 775 776 777 778 779 780 781 782 783 784 785 786 787 788 789 790 791 792 793 794 795 796 797 798 799 800 801 802 803 804 805 806 807 808 809 810 811 812 813 814 815 816 817 818 819 820 821 822 823 824 825 826 827 828 829 830 831 832 833 834 835 836 837 838 839 840 841 842 843 844 845 846 847 848 849 850 851 852 853 854 855 856 857 858 859 860 861 862 863 864 865 866 867 868 869 870 871 872 873 874 875 876 877 878 879 880 881 882 883 884 885 886 887 888 889 890 891 892 893 894 895 896 897 898 899 900 901 902 903 904 905 906 907 908 909 910 911 912 913 914 915 916 917 918 919 920 921 922 923 924 925 926 927 928 929 930 931 932 933 934 935 936 937 938 939 940 941 942 943 944 945 946 947 948 949 950 951 952 953 954 955 956 957 958 959 960 961 962 963 964 965 966 967 968 969 970 971 972 973 974 975 976 977 978 979 980 981 982 983 984 985 986 987 988 989 990 991 992 993 994 995 996 997 998 999 1000

4-6 NettyTimeClientHandler

```

14    public class TimeClientHandler extends
ChannelHandlerAdapter {
15
16        private static final Logger logger = Logger
17            .getLogger(TimeClientHandler.class.getName());
18
19        private int counter;
20
21        private byte[] req;
22
23        /**
24         * Creates a client-side handler.
25         */
26        public TimeClientHandler() {
27            req = ("QUERY TIME ORDER" +
System.getProperty("line.separator"))

```

```

28    .getBytes();
29    }
30
31    @Override
32    public void channelActive(ChannelHandlerContext ctx)
{
33    ByteBuf message = null;
34    for (int i = 0; i < 100; i++) {
35        message = Unpooled.buffer(req.length);
36        message.writeBytes(req);
37        ctx.writeAndFlush(message);
38    }
39    }
40
41    @Override
42    public void channelRead(ChannelHandlerContext ctx,
Object msg)
43        throws Exception {
44        String body = (String) msg;

```

```

45    System.out.println("Now is : " + body + " ; the counter
is : "

```

```
46    + ++counter);
```

```
47    }
```

```
48
```

```
49    @Override
```

```
50    public void exceptionCaught(ChannelHandlerContext  
ctx, Throwable cause) {
```

```
51    // 
```

```
52    logger.warning("Unexpected exception from downstream : "
```

```
53    + cause.getMessage());
```

```
54    ctx.close();
```

```
55    }
```

```
56 }
```

```
44 46 msg
```

4.3.3 TCP

TimeServerTimeClient

The time server receive order : QUERY TIME ORDER ; the
counter is : 1

The time server receive order : QUERY TIME ORDER ; the
counter is : 2

The time server receive order : QUERY TIME ORDER ; the
counter is : 3

The time server receive order : QUERY TIME ORDER ; the
counter is : 4

The time server receive order : QUERY TIME ORDER ; the
counter is : 5

The time server receive order : QUERY TIME ORDER ; the
counter is : 6

The time server receive order : QUERY TIME ORDER ; the
counter is : 7

The time server receive order : QUERY TIME ORDER ; the
counter is : 8

The time server receive order : QUERY TIME ORDER ; the
counter is : 9

The time server receive order : QUERY TIME ORDER ; the
counter is : 10

The time server receive order : QUERY TIME ORDER ; the
counter is : 11

The time server receive order : QUERY TIME ORDER ; the

counter is : 12

The time server receive order : QUERY TIME ORDER ; the
counter is : 13

The time server receive order : QUERY TIME ORDER ; the
counter is : 14

The time server receive order : QUERY TIME ORDER ; the
counter is : 15

The time server receive order : QUERY TIME ORDER ; the
counter is : 16

The time server receive order : QUERY TIME ORDER ; the
counter is : 17

The time server receive order : QUERY TIME ORDER ; the
counter is : 18

The time server receive order : QUERY TIME ORDER ; the
counter is : 19

The time server receive order : QUERY TIME ORDER ; the
counter is : 20

The time server receive order : QUERY TIME ORDER ; the
counter is : 21

The time server receive order : QUERY TIME ORDER ; the
counter is : 22

The time server receive order : QUERY TIME ORDER ; the
counter is : 23

The time server receive order : QUERY TIME ORDER ; the
counter is : 24

The time server receive order : QUERY TIME ORDER ; the
counter is : 25

The time server receive order : QUERY TIME ORDER ; the counter is : 26

The time server receive order : QUERY TIME ORDER ; the counter is : 27

The time server receive order : QUERY TIME ORDER ; the counter is : 28

The time server receive order : QUERY TIME ORDER ; the counter is : 29

The time server receive order : QUERY TIME ORDER ; the counter is : 30

The time server receive order : QUERY TIME ORDER ; the counter is : 31

The time server receive order : QUERY TIME ORDER ; the counter is : 32

The time server receive order : QUERY TIME ORDER ; the counter is : 33

The time server receive order : QUERY TIME ORDER ; the counter is : 34

The time server receive order : QUERY TIME ORDER ; the counter is : 35

The time server receive order : QUERY TIME ORDER ; the counter is : 36

The time server receive order : QUERY TIME ORDER ; the counter is : 37

The time server receive order : QUERY TIME ORDER ; the counter is : 38

The time server receive order : QUERY TIME ORDER ; the

counter is : 39

The time server receive order : QUERY TIME ORDER ; the
counter is : 40

The time server receive order : QUERY TIME ORDER ; the
counter is : 41

The time server receive order : QUERY TIME ORDER ; the
counter is : 42

The time server receive order : QUERY TIME ORDER ; the
counter is : 43

The time server receive order : QUERY TIME ORDER ; the
counter is : 44

The time server receive order : QUERY TIME ORDER ; the
counter is : 45

The time server receive order : QUERY TIME ORDER ; the
counter is : 46

The time server receive order : QUERY TIME ORDER ; the
counter is : 47

The time server receive order : QUERY TIME ORDER ; the
counter is : 48

The time server receive order : QUERY TIME ORDER ; the
counter is : 49

The time server receive order : QUERY TIME ORDER ; the
counter is : 50

The time server receive order : QUERY TIME ORDER ; the
counter is : 51

The time server receive order : QUERY TIME ORDER ; the
counter is : 52

The time server receive order : QUERY TIME ORDER ; the counter is : 53

The time server receive order : QUERY TIME ORDER ; the counter is : 54

The time server receive order : QUERY TIME ORDER ; the counter is : 55

The time server receive order : QUERY TIME ORDER ; the counter is : 56

The time server receive order : QUERY TIME ORDER ; the counter is : 57

The time server receive order : QUERY TIME ORDER ; the counter is : 58

The time server receive order : QUERY TIME ORDER ; the counter is : 59

The time server receive order : QUERY TIME ORDER ; the counter is : 60

The time server receive order : QUERY TIME ORDER ; the counter is : 61

The time server receive order : QUERY TIME ORDER ; the counter is : 62

The time server receive order : QUERY TIME ORDER ; the counter is : 63

The time server receive order : QUERY TIME ORDER ; the counter is : 64

The time server receive order : QUERY TIME ORDER ; the counter is : 65

The time server receive order : QUERY TIME ORDER ; the

counter is : 66

The time server receive order : QUERY TIME ORDER ; the
counter is : 67

The time server receive order : QUERY TIME ORDER ; the
counter is : 68

The time server receive order : QUERY TIME ORDER ; the
counter is : 69

The time server receive order : QUERY TIME ORDER ; the
counter is : 70

The time server receive order : QUERY TIME ORDER ; the
counter is : 71

The time server receive order : QUERY TIME ORDER ; the
counter is : 72

The time server receive order : QUERY TIME ORDER ; the
counter is : 73

The time server receive order : QUERY TIME ORDER ; the
counter is : 74

The time server receive order : QUERY TIME ORDER ; the
counter is : 75

The time server receive order : QUERY TIME ORDER ; the
counter is : 76

The time server receive order : QUERY TIME ORDER ; the
counter is : 77

The time server receive order : QUERY TIME ORDER ; the
counter is : 78

The time server receive order : QUERY TIME ORDER ; the
counter is : 79

The time server receive order : QUERY TIME ORDER ; the counter is : 80

The time server receive order : QUERY TIME ORDER ; the counter is : 81

The time server receive order : QUERY TIME ORDER ; the counter is : 82

The time server receive order : QUERY TIME ORDER ; the counter is : 83

The time server receive order : QUERY TIME ORDER ; the counter is : 84

The time server receive order : QUERY TIME ORDER ; the counter is : 85

The time server receive order : QUERY TIME ORDER ; the counter is : 86

The time server receive order : QUERY TIME ORDER ; the counter is : 87

The time server receive order : QUERY TIME ORDER ; the counter is : 88

The time server receive order : QUERY TIME ORDER ; the counter is : 89

The time server receive order : QUERY TIME ORDER ; the counter is : 90

The time server receive order : QUERY TIME ORDER ; the counter is : 91

The time server receive order : QUERY TIME ORDER ; the counter is : 92

The time server receive order : QUERY TIME ORDER ; the

counter is : 93

The time server receive order : QUERY TIME ORDER ; the
counter is : 94

The time server receive order : QUERY TIME ORDER ; the
counter is : 95

The time server receive order : QUERY TIME ORDER ; the
counter is : 96

The time server receive order : QUERY TIME ORDER ; the
counter is : 97

The time server receive order : QUERY TIME ORDER ; the
counter is : 98

The time server receive order : QUERY TIME ORDER ; the
counter is : 99

The time server receive order : QUERY TIME ORDER ; the
counter is : 100

□□□□□□□□

Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 1

Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 2

Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 3

Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 4

Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 5

Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 6

Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 7

Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 8

Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 9
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 10
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 11
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 12
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 13
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 14
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 15
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 16
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 17
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 18
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 19
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 20
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 21
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 22
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 23
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 24
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 25
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 26
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 27
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 28
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 29
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 30
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 31
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 32
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 33
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 34
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 35

Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 36
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 37
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 38
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 39
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 40
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 41
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 42
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 43
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 44
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 45
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 46
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 47
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 48
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 49
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 50
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 51
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 52
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 53
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 54
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 55
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 56
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 57
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 58
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 59
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 60
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 61
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 62

Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 63
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 64
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 65
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 66
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 67
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 68
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 69
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 70
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 71
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 72
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 73
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 74
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 75
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 76
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 77
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 78
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 79
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 80
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 81
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 82
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 83
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 84
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 85
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 86
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 87
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 88
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 89

```
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 90
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 91
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 92
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 93
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 94
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 95
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 96
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 97
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 98
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is : 99
Now is : Thu Feb 20 00:00:14 CST 2014 ; the counter is :
```

100

```
LineBasedFrameDecoder
StringDecoder
TCP
handler
ChannelPipeline
```

```
LineBasedFrameDecoder
StringDecoder
TCP
```

4.3.4 LineBasedFrameDecoderStringDecoder

```
LineBasedFrameDecoder
ByteBuf
"\n"
"\r\n"
"
```

StringDecoder 和 LineBasedFrameDecoder + StringDecoder 是 TCP 的常用组合

Netty 的 TCP 和 / 的常用组合

5 个常用组合

4.4 总结

TCP 的常用组合 3 个常用组合 TCP 和 / 的常用组合 LineBasedFrameDecoder+StringDecoder 是 TCP 的常用组合

5 网络编程

TCP 4

1 LEN

2 FTP

3

4

Netty 4 TCP

4 LineBasedFrameDecoder TCP
——DelimiterBasedFrameDecoder
FixedLengthFrameDecoder TCP/

- DelimiterBasedFrameDecoder
- DelimiterBasedFrameDecoder
- DelimiterBasedFrameDecoder
- FixedLengthFrameDecoder
- telnet FixedLengthFrameDecoder

5.1 DelimiterBasedFrameDecoder

DelimiterBasedFrameDecoder는 프레임의 길이를 지정하여 데이터를 읽어들이는 데 사용됩니다. DelimiterBasedFrameDecoder는 프레임의 길이를 지정하여 데이터를 읽어들이는 데 사용됩니다.

EchoServer는 EchoClient와 연결되어 데이터를 읽어들이고 "\$_"로 데이터를 출력합니다.

5.1.1 DelimiterBasedFrameDecoder

EchoServer는 다음과 같습니다.

5-1 EchoServer

```
22. public class EchoServer {
23.     public void bind(int port) throws Exception {
24.         // NIO
25.         EventLoopGroup bossGroup = new
NioEventLoopGroup();
26.         EventLoopGroup workerGroup = new
NioEventLoopGroup();
27.         try {
28.             ServerBootstrap b = new ServerBootstrap();
29.             b.group(bossGroup, workerGroup)
30.             .channel(NioServerSocketChannel.class)
31.             .option(ChannelOption.SO_BACKLOG, 100)
```



```

32.                                     .handler(new
LoggingHandler(LogLevel.INFO))
33.                                     .childHandler(new
ChannelInitializer<SocketChannel>() {
34.                                     @Override
35.                                     public void initChannel(SocketChannel ch)
36.                                     throws Exception {
37.                                     ByteBuf delimiter =
Unpooled.copiedBuffer("$_"

```

```

38.                                     .getBytes());

```

```

39.                                     ch.pipeline().addLast(

```

```

40.                                     new

```

```
DelimiterBasedFrameDecoder(1024,
```

```
41.                                     delimiter));
```

```
42. ch.pipeline().addLast(new  
StringDecoder());
```

```
43. ch.pipeline().addLast(new  
EchoServerHandler());
```

44. }

45. `});`

46.

47. //

```
48.         ChannelFuture f = b.bind(port).sync();
```

49.

50. //

```
51.         f.channel().closeFuture().sync();
```

```
52.         } finally {
```

53. //

```
54.         bossGroup.shutdownGracefully();
```

```

55.         workerGroup.shutdownGracefully();
56.     }
57. }
58.
59.     public static void main(String[] args) throws
Exception {
60.         int port = 8080;
61.         if (args != null && args.length > 0) {
62.             try {
63.                 port = Integer.valueOf(args[0]);
64.             } catch (NumberFormatException e) {
65.                 // 忽略
66.             }
67.         }
68.         new EchoServer().bind(port);
69.     }
70. }

```

37 41 ByteBuffer "\$_"
 40 DelimiterBasedFrameDecoder
 ChannelPipeline DelimiterBasedFrameDecoder
 1024
 TooLongFrame Exception
 Netty

EchoServerHandler

5-2 EchoServerEchoServerHandler

```
13.  @Sharable
      14.      public class EchoServerHandler extends
ChannelHandlerAdapter {
15.
16.      int counter = 0;
17.
18.      @Override
      19.      public void channelRead(ChannelHandlerContext
ctx, Object msg)
20.          throws Exception {
21.          String body = (String) msg;

      22.          System.out.println("This is " + ++counter + "
times receive client : ["

23.          + body + "]);
```

```
24.         body += "$_";
```

```
25.                                     ByteBuf echo =  
Unpooled.copiedBuffer(body.getBytes());
```

```
26.         ctx.writeAndFlush(echo);
```

```
27.     }
```

```
28.
```

```
29.     @Override
```

```

30.         public void exceptionCaught(ChannelHandlerContext
ctx, Throwable cause) {
31.             cause.printStackTrace();
32.             ctx.close(); // 關閉通道
33.         }
34.     }

```

21~23 行分別宣告了 `DelimiterBasedFrameDecoder`、`ChannelHandler` 以及 `msg` 變量。24 行宣告了 `ChannelHandler` 的 `StringDecoder` 以及 `ByteBuf` 變量。25 行宣告了 `EchoServerHandler` 以及 `msg` 變量。

26 行宣告了 `DelimiterBasedFrameDecoder` 變量。27 行宣告了 `ByteBuf` 變量。

28 行宣告了 `msg` 變量。

5.1.2 `DelimiterBasedFrameDecoder`

29 行宣告了 `EchoClient` 變量。

30 行宣告了 `EchoClient` 變量。

```

20.     public class EchoClient {
21.
22.         public void connect(int port, String host) throws
Exception {
23.             // 宣告 NIO

```

```

24.      EventLoopGroup group = new NioEventLoopGroup();
25.      try {
26.          Bootstrap b = new Bootstrap();
27.
b.group(group).channel(NioSocketChannel.class)
28.          .option(ChannelOption.TCP_NODELAY, true)
29.          .handler(new
ChannelInitializer<SocketChannel>() {
30.              @Override
31.              public void initChannel(SocketChannel ch)
32.                  throws Exception {
33.                  ByteBuf delimiter =
Unpooled.copiedBuffer("$_"
34.                      .getBytes());

35.                  ch.pipeline().addLast(

```

```
36.                                                                 new  
DelimiterBasedFrameDecoder(1024,
```

```
37.                                                                 delimiter));
```

```
38.                                                                 ch.pipeline().addLast(new  
StringDecoder());
```

```
39.                                                                 ch.pipeline().addLast(new  
EchoClientHandler());
```



```

40.         }
41.     });
42.
43.     // 關閉通道
44.     ChannelFuture f = b.connect(host,
port).sync();
45.
46.     // 關閉通道
47.     f.channel().closeFuture().sync();
48. } finally {
49.     // 關閉NIO通道
50.     group.shutdownGracefully();
51. }
52. }
53.
54. /**
55.  * @param args
56.  * @throws Exception
57.  */
58.     public static void main(String[] args) throws
Exception {
59.         int port = 8080;
60.         if (args != null && args.length > 0) {
61.             try {

```

```

62.         port = Integer.valueOf(args[0]);
63.     } catch (NumberFormatException e) {
64.         // 忽略
65.     }
66. }
67. new EchoClient().connect(port, "127.0.0.1");
68. }
69. }

```

在 `DelimiterBasedFrameDecoder` 和 `StringDecoder` 之间添加 `ChannelPipeline` 和 `I/O` 线程的 `EchoClientHandler` 和 `EchoClientHandler`。

图 5-4 EchoClient 和 EchoClientHandler

```

11.     public class EchoClientHandler extends
ChannelHandlerAdapter {
12.
13.         private int counter;
14.
15.         static final String ECHO_REQ = "Hi, Lilinfeng.
Welcome to Netty.$_";
16.
17.         /**
18.          * Creates a client-side handler.
19.          */

```

```
20.     public EchoClientHandler() {
21.     }
22.
23.     @Override
24.     public void channelActive(ChannelHandlerContext
ctx) {
25.         for (int i = 0; i < 10; i++) {
ctx.writeAndFlush(Unpooled.copiedBuffer(ECHO_REQ.getBytes()));

26.     }

27. }

28.
29.     @Override
```

```

30.      public void channelRead(ChannelHandlerContext
ctx, Object msg)
31.      throws Exception {
32.      System.out.println("This is " + ++counter + "
times receive server : ["

33.      + msg + "]");

34.      }
35.
36.      @Override
37.      public void
channelReadComplete(ChannelHandlerContext ctx) throws Exception
{
38.      ctx.flush();
39.      }
40.
41.      @Override
42.      public void exceptionCaught(ChannelHandlerContext

```

```
ctx, Throwable cause) {  
    43.         cause.printStackTrace();  
    44.         ctx.close();  
    45.     }  
    46. }
```

25 26 TCP 32 33

5.1.3 DelimiterBasedFrameDecoder

This is 1 times receive client : [Hi, Lilinfeng. Welcome to Netty.]

This is 2 times receive client : [Hi, Lilinfeng. Welcome to Netty.]

This is 3 times receive client : [Hi, Lilinfeng. Welcome to Netty.]

This is 4 times receive client : [Hi, Lilinfeng. Welcome to Netty.]

This is 5 times receive client : [Hi, Lilinfeng. Welcome to Netty.]

This is 6 times receive client : [Hi, Lilinfeng. Welcome to Netty.]

This is 7 times receive client : [Hi, Lilinfeng. Welcome to Netty.]

This is 8 times receive client : [Hi, Lilinfeng. Welcome to Netty.]

This is 9 times receive client : [Hi, Lilinfeng. Welcome to Netty.]

This is 10 times receive client : [Hi, Lilinfeng. Welcome to Netty.]

□□□□□□□□

This is 1 times receive server : [Hi, Lilinfeng. Welcome to Netty.]

This is 2 times receive server : [Hi, Lilinfeng. Welcome to Netty.]

This is 3 times receive server : [Hi, Lilinfeng. Welcome to Netty.]

This is 4 times receive server : [Hi, Lilinfeng. Welcome to Netty.]

This is 5 times receive server : [Hi, Lilinfeng. Welcome to Netty.]

This is 6 times receive server : [Hi, Lilinfeng. Welcome to Netty.]

This is 7 times receive server : [Hi, Lilinfeng. Welcome to Netty.]

This is 8 times receive server : [Hi, Lilinfeng. Welcome to

ByteBuffer 的 getChar 方法。TCP 的
ByteBuffer

5.2 FixedLengthFrameDecoder

FixedLengthFrameDecoder 是一个解码器，它用于解码固定长度的帧。它通常用于 TCP 连接，其中每个帧的长度是固定的。

5.2.1 FixedLengthFrameDecoder

在 ChannelPipeline 中添加 FixedLengthFrameDecoder 的步骤如下。在 20 行代码中，我们添加了一个 EchoServerHandler。

图 5-5 EchoServer 的 EchoServer

```
20. public class EchoServer {
21.     public void bind(int port) throws Exception {
22.         // 添加 NIO 的
23.         EventLoopGroup bossGroup = new
NioEventLoopGroup();
24.         EventLoopGroup workerGroup = new
NioEventLoopGroup();
25.         try {
26.             ServerBootstrap b = new ServerBootstrap();
27.             b.group(bossGroup, workerGroup)
28.             .channel(NioServerSocketChannel.class)
29.             .option(ChannelOption.SO_BACKLOG, 100)
30.             .handler(new
```



```

LoggingHandler(LogLevel.INFO))
    31.                .childHandler(new
ChannelInitializer<SocketChannel>() {
    32.                @Override
    33.                public void initChannel(SocketChannel ch)
    34.                throws Exception {
    35.                ch.pipeline().addLast(

    36.                new FixedLengthFrameDecoder(20));

    37.                ch.pipeline().addLast(new
StringDecoder());
    38.                ch.pipeline().addLast(new
EchoServerHandler());
    39.                }
    40.                });
    41.
    42.                // 000000000000
    43.                ChannelFuture f = b.bind(port).sync();

```

```

44.
45.         // 關閉通道
46.         f.channel().closeFuture().sync();
47.     } finally {
48.         // 關閉 BossWorker
49.         bossGroup.shutdownGracefully();
50.         workerGroup.shutdownGracefully();
51.     }
52. }
53.
54.     public static void main(String[] args) throws
Exception {
55.         int port = 8080;
56.         if (args != null && args.length > 0) {
57.             try {
58.                 port = Integer.valueOf(args[0]);
59.             } catch (NumberFormatException e) {
60.                 // 錯誤
61.             }
62.         }
63.         new EchoServer().bind(port);
64.     }
65. }

```

EchoServerHandler 的實現

圖 5-6 EchoServer 的 EchoServerHandler

```
11.  @Sharable
12.      public class EchoServerHandler extends
ChannelHandlerAdapter {
13.
14.      @Override
15.      public void channelRead(ChannelHandlerContext
ctx, Object msg)
16.          throws Exception {
17.          System.out.println("Receive client : [" + msg +
"]");
18.      }
19.
20.      @Override
21.      public void exceptionCaught(ChannelHandlerContext
ctx, Throwable cause) {
22.          cause.printStackTrace();
23.          ctx.close();// 关闭连接
24.      }
25.  }
```

FixedLengthFrameDecoder 是 Netty 提供的一个解码器，它用于解码固定长度的帧。它实现 ChannelHandler 接口，并实现 channelRead 方法。在 channelRead 方法中，它调用 decode 方法，将接收到的数据解码为一个字节数组。如果解码成功，它将数据放入一个 ByteBuffer 中，并返回给下一个处理器。如果解码失败，它将数据放入一个 ByteBuffer 中，并返回给下一个处理器。FixedLengthFrameDecoder 的构造函数接受一个整数，表示帧的长度。如果帧的长度不是 4 的倍数，它将抛出 IllegalArgumentException。

telnet 客户端连接到 EchoServer，并发送数据。

5.2.2 使用telnet测试EchoServer

使用telnet测试EchoServer

在Windows的命令提示符中输入telnet，按回车键，显示如下：

```
Lilinfeng welcome to Netty at Nanjing
```

显示“Lilinfeng welcome to”

按回车键，显示如下：

1. 在cmd中输入5-2

5-2 在cmd中输入CMD

2. 在telnet中输入“telnet localhost 8080”，按回车键，显示如下：

5-3 在telnet中输入

3. 在set localecho中输入5-4

5-4 显示Lilinfeng welcome to Netty at Nanjing

4. EchoServer显示5-5

5-5 显示结果

图5-5 固定长度帧解码器 FixedLengthFrameDecoder
接收20字节的帧，解码后得到“Lilinfeng welcome to”

5.3 帧解码器

帧解码器是 Netty 框架中用于解码帧的组件。Netty 框架提供了两种帧解码器：FixedLengthFrameDecoder 和 DelimiterBasedFrameDecoder。

FixedLengthFrameDecoder 用于解码固定长度的帧。它接收一个字节数组，并返回一个解码后的字符串。它适用于 TCP 连接，因为 TCP 连接中的数据流是连续的，不需要分隔符来区分帧。

DelimiterBasedFrameDecoder 用于解码带有分隔符的帧。它接收一个字节数组，并返回一个解码后的字符串。它适用于 HTTP 连接，因为 HTTP 连接中的数据流是由分隔符分隔的。Netty 框架还提供了 ChannelPipeline 类，用于管理解码器的生命周期。

Netty 框架还提供了 NIO 接口，用于处理网络 I/O 操作。Netty 框架还支持 Java 的 protobuf 和 JBoss 的 Marshalling 接口，用于处理序列化数据。

□□□ **Netty**□□□□□□□□

□6□ □□□□□

□7□ Java□□□

□8□ Google Protobuf□□□

□9□ JBoss Marshalling□□□

第6章 网络编程

本章介绍 Java 网络编程 API / 包。ObjectInputStream 和 ObjectOutputStream 是 Java 序列化 API 的重要组成部分。JDK 提供了多种网络编程 API。

Java 网络编程 API

- 网络编程
- 网络通信

本章介绍 Netty 和 NIO 网络编程 API。ByteBuffer 是 Java 网络编程 API 的重要组成部分。ByteBuffer 是 Java 网络编程 API 的重要组成部分。

Java 网络编程 API 是 Java 网络编程 API 的重要组成部分。Netty 是 Java 网络编程 API 的重要组成部分。

网络编程 API

- Java 网络编程 API
- 网络通信 API

6.1 Java 网络编程 API

Java 网络编程 API 是 JDK 1.1 的重要组成部分。java.io.Serializable 是 Java 网络编程 API 的重要组成部分。

RPCJava
Java

6.1.1

Java C++
Java

Java
Java

Java RCP
RPC

6.1.2

Java

6-1 Java POJO UserInfo

```
10. public class UserInfo implements Serializable {
11.
12.     /**
13.      *
14.      */
15.     private static final long serialVersionUID = 1L;
16.
17.     private String userName;
18.
```

```
19.     private int userID;
20.
21.     public UserInfo buildUserName(String userName) {
22.         this.userName = userName;
23.         return this;
24.     }
25.
26.     public UserInfo buildUserID(int userID) {
27.         this.userID = userID;
28.         return this;
29.     }
30.
31.     /**
32.      * @return the userName
33.      */
34.     public final String getUserName() {
35.         return userName;
36.     }
37.
38.     /**
39.      * @param userName
40.      *         the userName to set
41.      */
42.     public final void setUserName(String userName) {
43.         this.userName = userName;
44.     }
45.
```

```
46.      /**
47.         * @return the userID
48.         */
49.     public final int getUserID() {
50.         return userID;
51.     }
52.
53.     /**
54.         * @param userID
55.         *           the userID to set
56.         */
57.     public final void setUserID(int userID) {
58.         this.userID = userID;
59.     }
60.
61.     public byte[] codeC() {
62.         ByteBuffer buffer = ByteBuffer.allocate(1024);
63.         byte[] value = this.userName.getBytes();
64.         buffer.putInt(value.length);
65.         buffer.put(value);
66.         buffer.putInt(this.userID);
67.         buffer.flip();
68.         value = null;
69.         byte[] result = new byte[buffer.remaining()];
70.         buffer.get(result);
71.         return result;
```

```
72.         }
```

```
73.     }
```

```
    UserInfo implements POJO implements java.io.Serializable {
        private static final long serialVersionUID = 1L;
        private UserInfo() {}
        // ...
    }
```

```
    // 61-72: ByteBuffer to UserInfo
    private byte[] toByteArray() {
        // ...
    }
```

```
    // ...
    // ...
    // ...
}
```

6-2 Java TestUserInfo

```
11. public class TestUserInfo {
12.
13.     /**
14.      * @param args
15.      * @throws IOException
16.      */
17.     public static void main(String[] args) throws
IOException {
18.         UserInfo info = new UserInfo();
19.         info.buildUserID(100).buildUserName("Welcome to
Netty");
20.         ByteArrayOutputStream bos = new
```

```

ByteArrayOutputStream();
    21.                ObjectOutputStream  os  =  new
ObjectOutputStream(bos);
    22.        os.writeObject(info);
    23.        os.flush();
    24.        os.close();
    25.        byte[] b = bos.toByteArray();
    26.        System.out.println("The jdk serializable length
is : " + b.length);
    27.        bos.close();
    28.        System.out.println("-----
-----");
    29.        System.out.println("The byte array serializable
length is : "
    30.            + info.getCodeC().length);
    31.    }
    32.
    33. }

```

-


```

IOException {
    19.      UserInfo info = new UserInfo();
    20.      info.buildUserID(100).buildUserName("Welcome to
Netty");
    21.      int loop = 1000000;
    22.      ByteArrayOutputStream bos = null;
    23.      ObjectOutputStream os = null;
    24.      long startTime = System.currentTimeMillis();
    25.      for (int i = 0; i < loop; i++) {
    26.          bos = new ByteArrayOutputStream();
    27.          os = new ObjectOutputStream(bos);
    28.          os.writeObject(info);
    29.          os.flush();
    30.          os.close();
    31.          byte[] b = bos.toByteArray();
    32.          bos.close();
    33.      }
    34.      long endTime = System.currentTimeMillis();
    35.      System.out.println("The jdk serializable cost
time is : "
    36.          + (endTime - startTime) + " ms");
    37.
    38.      System.out.println("-----
-----");
    39.
    40.      ByteBuffer buffer = ByteBuffer.allocate(1024);
    41.      startTime = System.currentTimeMillis();

```



```

42.         for (int i = 0; i < loop; i++) {
43.             byte[] b = info.codeC(buffer);
44.         }
45.         endTime = System.currentTimeMillis();
46.         System.out.println("The byte array serializable
cost time is : "
47.             + (endTime - startTime) + " ms");
48.     }
49. }

```

Java 100
6-3

6-3 UserInfo

Java 6.17% Java

Java 6-4

6-4

6-4 JDK Java

JDK JDK

6.2 序列化与反序列化

Java 序列化与反序列化是 Java 中非常基础且重要的概念，用于在内存中的对象与字节流之间进行转换。本文将介绍 Java 序列化与反序列化的基本原理、使用场景以及相关的 API。

6.2.1 Google Protobuf

Protobuf (Google Protocol Buffers) 是一种轻量级的、高效的序列化格式。它允许开发者定义自己的数据类型，并将这些类型的数据序列化为紧凑的二进制格式。Protobuf 支持多种语言，包括 Java、C++、Python 等。本文将介绍 Protobuf 的基本概念和使用方法。

序列化与反序列化

- 序列化：将内存中的对象转换为字节流的过程。
- 反序列化：将字节流转换回内存中的对象的过程。
- 序列化与反序列化的重要性：在分布式系统、网络通信、数据存储等场景中，数据需要在不同节点之间传输，序列化与反序列化是实现这一过程的关键。
- 支持的语言：Java、C++、Python 等。

Protobuf 与 XML 和 JSON 的对比。XML 和 JSON 是常见的序列化格式，但它们存在一些问题，如 XML 的冗余性、JSON 的灵活性不足等。Protobuf 则通过其紧凑的二进制格式和强大的类型系统，解决了这些问题。本文将对比 Protobuf 与 XML 和 JSON 的优缺点，并介绍 Protobuf 的基本使用。

Protobuf 的基本使用。本文将介绍 Protobuf 的基本使用，包括如何定义数据类型、如何序列化与反序列化数据。我们将通过一个简单的例子来说明 Protobuf 的使用。

- 定义数据类型：使用 .proto 文件定义数据类型。
- 序列化与反序列化：使用 Protobuf 提供的 API 进行序列化与反序列化。
- 编译与运行：使用 Protobuf 编译器生成目标语言代码。
- 性能对比：Protobuf 与 XML 和 JSON 的性能对比。

Protobuf 6-5 6-6

6-5 Protobuf

6-6 Protobuf

6-5 6-6 Protobuf
RPC Protobuf

6.2.2 Facebook Thrift

Thrift Facebook 2007 Facebook Thrift
Apache Facebook Thrift Facebook
Thrift
C++ C# Cocoa Erlang Haskell Java Ocaml Perl PHP
Python Ruby Smalltalk

Thrift
RPC Thrift
IDL IDL Thrift Thrift
JSON XML

Thrift 5

1 IDL IDL

2 TProtocol RPC JSON
Binary

3 TTransport RPC socket
NIO MemoryBuffer

4 TProcessor

5 TServer TProtocol TTransport TProcessor

TProtocol Thrift RPC
Thrift RPC TProtocol
Thrift

Protobuf Thrift IDL 8 Java
Map Set List

Thrift

-
-
-

Thrift Java RMI
6-7

6-7 Thrift

6.2.3 JBoss Marshalling

JBoss Marshalling Java API JDK
java.io.Serializable

序列化与反序列化

序列化Java对象

- 序列化与反序列化
- 序列化与反序列化
- 序列化与反序列化
- 实现java.io.Serializable接口
- 序列化与反序列化

序列化与反序列化JBoss Marshalling与JBoss序列化

JBoss Marshalling与Netty Marshalling

6.3 序列化

序列化Java对象Java序列化与反序列化
序列化与反序列化7序列化与反序列化Netty
序列化与反序列化

第7章 Java序列化

本章主要介绍Java序列化技术，包括Java序列化接口、POJO、java.io.Serializable、ID、java.io.ObjectInput、java.io.ObjectOutput等。

本章主要介绍Java序列化技术，包括Java序列化接口、POJO、java.io.Serializable、ID、java.io.ObjectInput、java.io.ObjectOutput等。

本章主要介绍Java序列化技术，包括Java序列化接口、POJO、java.io.Serializable、ID、java.io.ObjectInput、java.io.ObjectOutput等。

- Netty Java序列化接口
- Netty Java序列化接口
- Netty Java序列化接口

7.1 Netty Java序列化接口

本章主要介绍Netty序列化接口，包括7-1、7-2等。

7-1 SubscribeReq

本章主要介绍Netty序列化接口，包括7-2等。

7-2 SubscribeResp

本章主要介绍Netty序列化接口，包括ObjectEncoder、ObjectDecoder等。

本章主要介绍Netty序列化接口，包括ObjectEncoder、ObjectDecoder等。

Netty POJO

1 ChannelPipeline io.netty.handler.codec.serialization.Object Decoder

2 ChannelPipeline io.netty.handler.codec.serialization.Object Encoder

3 Java POJO java.io.Serializable

Netty POJO Java

7-1 Netty Java POJO

```
9. public class SubscribeReq implements Serializable {
10.
11.     /**
12.      * ID
13.      */
14.     private static final long serialVersionUID = 1L;
15.
16.     private int subReqID;
17.
18.     private String userName;
19.
20.     private String productName;
21.
22.     private String phoneNumber;
```



```

23.
24.     private String address;
.....//get set
100.
101.     /*
102.     * (non-Javadoc)
103.     *
104.     * @see java.lang.Object#toString()
105.     */
106.     @Override
107.     public String toString() {
108.         return "SubscribeReq [subReqID=" + subReqID +
", userName=" + userName
109.         + ", productName=" + productName + ",
phoneNumber="
110.         + phoneNumber + ", address=" + address +
"]";
111.     }
112. }

```

SubscribeReq是一个JOJO对象

19个Serializable

214个ID

POJO

图7-2 Netty Java POJO

```
9.     public class SubscribeResp implements Serializable {
10.
11.         /**
12.          * 请求ID
13.          */
14.     private static final long serialVersionUID = 1L;
15.
16.     private int subReqID;
17.
18.     private int respCode;
19.
20.     private String desc;
    //get/set.....
66.
67.     /*
68.      * (non-Javadoc)
69.      *
70.      * @see java.lang.Object#toString()
71.      */
72.     @Override
73.     public String toString() {
74.         return "SubscribeResp [subReqID=" + subReqID + ",
respCode=" + respCode
75.             + ", desc=" + desc + "];"
```

```
76.     }
77. }
```

□□□□□□□□□□□□□□□□□□□□

□□□□7-3 Netty Java□□□ □□□□□□□SubReqServer

```
21.  public class SubReqServer {
22.      public void bind(int port) throws Exception {
23.          // □□□□□□NIO□□□
24.              EventLoopGroup bossGroup = new
NioEventLoopGroup();
25.              EventLoopGroup workerGroup = new
NioEventLoopGroup();
26.          try {
27.              ServerBootstrap b = new ServerBootstrap();
28.              b.group(bossGroup, workerGroup)
29.                  .channel(NioServerSocketChannel.class)
30.                  .option(ChannelOption.SO_BACKLOG, 100)
31.                  .handler(new
LoggingHandler(LogLevel.INFO))
32.                  .childHandler(new
ChannelInitializer<SocketChannel>() {
33.                      @Override
34.                      public void initChannel(SocketChannel ch)
{
```

35. `ch.pipeline()`

36. `.addLast()`

37. `new ObjectDecoder(`

38. `1024 * 1024,`

39. `ClassResolvers`

40.

.weakCachingConcurrentResolver(this

41.

.getClass()

42.

.getClassLoader())));

43.

*ch.pipeline().addLast(new
ObjectEncoder());*

```
44.                                     ch.pipeline().addLast(new
SubReqServerHandler());
```

```
45.                                     }
46.                                     });
47.
48.                                     // 關閉通道
49.                                     ChannelFuture f = b.bind(port).sync();
50.
51.                                     // 關閉通道
52.                                     f.channel().closeFuture().sync();
53.                                } finally {
54.                                    // 關閉通道
55.                                    bossGroup.shutdownGracefully();
56.                                    workerGroup.shutdownGracefully();
57.                                }
58.                                }
59.
60.                                public static void main(String[] args) throws
Exception {
```

```

61.         int port = 8080;
62.         if (args != null && args.length > 0) {
63.             try {
64.                 port = Integer.valueOf(args[0]);
65.             } catch (NumberFormatException e) {
66.                 // 忽略
67.             }
68.         }
69.         new SubReqServer().bind(port);
70.     }
71. }

```

35 添加 `ObjectDecoder` 实现 `Serializable` 和 `POJO` 的解码器，实现 `ClassResolver` 接口，使用 `weakCachingConcurrentResolver` 实现 `WeakReferenceMap`，实现 `1M` 的缓存。

43 添加 `ObjectEncoder` 实现 `Serializable` 和 `POJO` 的编码器，实现 `Netty` 的编码器。

44 添加 `handler` 实现 `SubReqServerHandler`，实现 `ChannelPipeline` 的 `SubReqServerHandler`。

例7-4 Netty Java 实现 SubReqServerHandler

```
14.  @Sharable
15.      public class SubReqServerHandler extends
ChannelHandlerAdapter {
16.
17.      @Override
18.          public void channelRead(ChannelHandlerContext
ctx, Object msg)
19.              throws Exception {
20.          SubscribeReq req = (SubscribeReq) msg;
21.                                  if
("Lilinfeng".equalsIgnoreCase(req.getUserName())) {
22.              System.out.println("Service accept client
subscribe req : ["
23.                  + req.toString() + "]);
24.              ctx.writeAndFlush(resp(req.getSubReqID()));
25.          }
26.      }
27.
28.      private SubscribeResp resp(int subReqID) {
29.          SubscribeResp resp = new SubscribeResp();
30.          resp.setSubReqID(subReqID);
31.          resp.setRespCode(0);
32.          resp.setDesc("Netty book order succeed, 3 days
later, sent to the designated address");
```



```

33.         return resp;
34.     }
35.
36.     @Override
37.     public void exceptionCaught(ChannelHandlerContext
ctx, Throwable cause) {
38.         cause.printStackTrace();
39.         ctx.close(); // 关闭通道
40.     }
41. }

```

在 handler ObjectDecoder 中，SubReqServerHandler
 接收 SubscribeReq 消息，并调用 21 号方法，
 将消息放入队列中。

代码如下：

7.2 Java 中 Netty 的应用

在 Netty 中，

1. 创建 Netty 的 ChannelPipeline

2. 创建 Netty 的 Java 的 TCP
 连接，并设置 10 秒的超时时间

3. 创建 handler

代码如下：

□□□□□□

□□□□7-5 Netty Java□□□ □□□□□□

```
19.    public class SubReqClient {
20.
21.        public void connect(int port, String host) throws
Exception {
22.            // □□□□□NIO□□□
23.            EventLoopGroup group = new NioEventLoopGroup();
24.            try {
25.                Bootstrap b = new Bootstrap();
26.
b.group(group).channel(NioSocketChannel.class)
27.                .option(ChannelOption.TCP_NODELAY, true)
28.                .handler(new
ChannelInitializer<SocketChannel>() {
29.                    @Override
30.                    public void initChannel(SocketChannel ch)
31.                        throws Exception {
32.                        ch.pipeline().addLast(
```

33.
ClassResolvers

new ObjectDecoder(1024,

.cacheDisabled(this.getClass()

34.

35.

.getClassLoader())));

36.
ObjectEncoder());

ch.pipeline().addLast(new

```
37.                                     ch.pipeline().addLast(new
SubReqClientHandler());
```

```
38.                                     }
39.                                     });
40.
41.                                     // 连接超时
42.                                     ChannelFuture f = b.connect(host,
port).sync();
43.
44.                                     // 关闭连接
45.                                     f.channel().closeFuture().sync();
46.     } finally {
47.         // 关闭NIO线程组
48.         group.shutdownGracefully();
49.     }
50. }
51.
52. /**
53.  * @param args
54.  * @throws Exception
55.  */
56.     public static void main(String[] args) throws
```

```

Exception {
    57.         int port = 8080;
    58.         if (args != null && args.length > 0) {
    59.             try {
    60.                 port = Integer.valueOf(args[0]);
    61.             } catch (NumberFormatException e) {
    62.                 // ignore
    63.             }
    64.         }
    65.         new SubReqClient().connect(port, "127.0.0.1");
    66.     }
    67. }

```

32

 bundle

 bundle

 bundle

SubReqClientHandler

7-6 Netty Java SubReqClientHandler

```

    12.         public class SubReqClientHandler extends
ChannelHandlerAdapter {
    13.
    14.             /**
    15.              * Creates a client-side handler.
    16.              */

```

```

17.         public SubReqClientHandler() {
18.         }
19.
20.         @Override
21.         public void channelActive(ChannelHandlerContext
ctx) {
22.             for (int i = 0; i < 10; i++) {
23.                 ctx.write(subReq(i));
24.             }
25.             ctx.flush();
26.         }
27.
28.         private SubscribeReq subReq(int i) {
29.             SubscribeReq req = new SubscribeReq();
30.             req.setAddress("XXXXXXXXXXXX");
31.             req.setPhoneNumber("138xxxxxxxx");
32.             req.setProductName("Netty XXXX");
33.             req.setSubReqID(i);
34.             req.setUsername("Lilinfeng");
35.             return req;
36.         }
37.
38.         @Override
39.         public void channelRead(ChannelHandlerContext
ctx, Object msg)
40.             throws Exception {
41.             System.out.println("Receive server response : ["

```

```

+ msg + "]);
42.    }
43.
44.    @Override
45.    public void
channelReadComplete(ChannelHandlerContext ctx) throws Exception
{
46.        ctx.flush();
47.    }
48.
49.    @Override
50.    public void exceptionCaught(ChannelHandlerContext
ctx, Throwable cause) {
51.        cause.printStackTrace();
52.        ctx.close();
53.    }
54. }

```

222510

SubReqClientHandler

7.3

Java

```
log4j:WARN No appenders could be found for logger
(io.netty.util.internal.logging.InternalLoggerFactory).
```

```
log4j:WARN Please initialize the log4j system properly.
```

```
Service accept client subscribe req : [SubscribeReq
[subReqID=0, userName= Lilinfeng, productName=Netty ,
phoneNumber=138xxxxxxxx, address=]]
```

```
Service accept client subscribe req : [SubscribeReq
[subReqID=1, userName= Lilinfeng, productName=Netty ,
phoneNumber=138xxxxxxxx, address=]]
```

```
Service accept client subscribe req : [SubscribeReq
[subReqID=2, userName= Lilinfeng, productName=Netty ,
phoneNumber=138xxxxxxxx, address=]]
```

```
Service accept client subscribe req : [SubscribeReq
[subReqID=3, userName= Lilinfeng, productName=Netty ,
phoneNumber=138xxxxxxxx, address=]]
```

```
Service accept client subscribe req : [SubscribeReq
[subReqID=4, userName= Lilinfeng, productName=Netty ,
phoneNumber=138xxxxxxxx, address=]]
```

```
Service accept client subscribe req : [SubscribeReq
[subReqID=5, userName= Lilinfeng, productName=Netty ,
phoneNumber=138xxxxxxxx, address=]]
```


Service accept client subscribe req : [SubscribeReq
[subReqID=6, userName=Lilinfeng, productName=Netty 书籍 ,
phoneNumber=138xxxxxxxx, address=北京市海淀区]]

Service accept client subscribe req : [SubscribeReq
[subReqID=7, userName= Lilinfeng, productName=Netty 书籍 ,
phoneNumber=138xxxxxxxx, address=北京市海淀区]]

Service accept client subscribe req : [SubscribeReq
[subReqID=8, userName= Lilinfeng, productName=Netty 书籍 ,
phoneNumber=138xxxxxxxx, address=北京市海淀区]]

Service accept client subscribe req : [SubscribeReq
[subReqID=9, userName= Lilinfeng, productName=Netty 书籍 ,
phoneNumber=138xxxxxxxx, address=北京市海淀区]]

10个TCP连接
10个连接

连接

log4j:WARN No appenders could be found for logger
(io.netty.util.internal.logging.InternalLoggerFactory).

log4j:WARN Please initialize the log4j system properly.

Receive server response : [SubscribeResp [subReqID=0,
respCode=0, desc=Netty book order succeed, 3 days later, sent
to the designated address]]

Receive server response : [SubscribeResp [subReqID=1,
respCode=0, desc=Netty book order succeed, 3 days later, sent

to the designated address]]

Receive server response : [SubscribeResp [subReqID=2, respCode=0, desc=Netty book order succeed, 3 days later, sent to the designated address]]

Receive server response : [SubscribeResp [subReqID=3, respCode=0, desc=Netty book order succeed, 3 days later, sent to the designated address]]

Receive server response : [SubscribeResp [subReqID=4, respCode=0, desc=Netty book order succeed, 3 days later, sent to the designated address]]

Receive server response : [SubscribeResp [subReqID=5, respCode=0, desc=Netty book order succeed, 3 days later, sent to the designated address]]

Receive server response : [SubscribeResp [subReqID=6, respCode=0, desc=Netty book order succeed, 3 days later, sent to the designated address]]

Receive server response : [SubscribeResp [subReqID=7, respCode=0, desc=Netty book order succeed, 3 days later, sent to the designated address]]

Receive server response : [SubscribeResp [subReqID=8, respCode=0, desc=Netty book order succeed, 3 days later, sent to the designated address]]

Receive server response : [SubscribeResp [subReqID=9, respCode=0, desc=Netty book order succeed, 3 days later, sent to the designated address]]

我们可以在10秒内完成Netty的ObjectEncoder和ObjectDecoder的测试。

7.4 测试

我们可以在Netty的ObjectEncoder和ObjectDecoder的测试中，使用POJO（Plain Old Java Object）来测试TCP连接/接收数据。

我们可以在Netty的Java handler中，使用POJO来测试handler的测试。

我们可以在Protobuf的Netty的Protobuf POJO中测试。

- Protobuf
- Netty
- Netty
- Netty

[illegible]

Protobuf POJO
Netty POJO Protobuf
POJO

8.1.1 Protobuf

Protobuf windows

<http://code.google.com/p/protobuf/downloads/detail?name=protoc-2.5.0-win32.zip&can=2&q=>

protoc-2.5.0-win32.zip 8-1

8-1 Protobuf

protoc.exe .proto 7
SubscribeReq.proto SubscribeResp.proto

- SubscribeReq.proto 8-2

8-2 SubscribeReq.proto

- SubscribeResp.proto 8-3

8-3 SubscribeResp.proto

protoc.exe Java 8-4.

8-4 protoc.exe

将 POJO 类 SubscribeReqProto.java 和
SubscribeRespProto.java 导入 Eclipse 中 8-5

图8-5 导入POJO类

将 protobuf-java-2.5.0.jar 添加到 Google 的
lib 中 8-6

图8-6 添加Protobuf

Protobuf 类

8.1.2 Protobuf

Protobuf 类 SubscribeReqProto

Protobuf

图8-1 Protobuf TestSubscribeReqProto

```
12. public class TestSubscribeReqProto {
13.
14.     private static byte[]
encode(SubscribeReqProto.SubscribeReq req) {
15.     return req.toByteArray();
16. }
17.
18.     private static SubscribeReqProto.SubscribeReq
```

```

decode(byte[] body)
    19.         throws InvalidProtocolBufferException {
                                   20.                 return
SubscribeReqProto.SubscribeReq.parseFrom(body);
    21.     }
    22.
    23.         private static SubscribeReqProto.SubscribeReq
createSubscribeReq() {
    24.             SubscribeReqProto.SubscribeReq.Builder builder =
SubscribeReqProto.SubscribeReq
    25.                 .newBuilder();
    26.             builder.setSubReqID(1);
    27.             builder.setUserName("Lilinfeng");
    28.             builder.setProductName("Netty Book");
    29.             List<String> address = new ArrayList<>();
    30.             address.add("NanJing YuHuaTai");
    31.             address.add("BeiJing LiuLiChang");
    32.             address.add("ShenZhen HongShuLin");
    33.             builder.addAllAddress(address);
    34.             return builder.build();
    35.         }
    36.
    37.         /**
    38.          * @param args
    39.          * @throws InvalidProtocolBufferException
    40.          */
    41.         public static void main(String[] args)

```



```

42.         throws InvalidProtocolBufferException {
43.             SubscribeReqProto.SubscribeReq req =
createSubscribeReq();
44.             System.out.println("Before encode : " +
req.toString());
45.             SubscribeReqProto.SubscribeReq req2 =
decode(encode(req));
46.             System.out.println("After decode : " +
req.toString());
47.             System.out.println("Assert equal : --> " +
req2.equals(req));
48.     }
49. }

```

SubscribeReqProto.SubscribeReq 24
 SubscribeReqProto.SubscribeReq newBuilder
 SubscribeReqProto.SubscribeReqBuilderBuilder
 SubscribeReqaddAllXXX()

SubscribeReqProto.SubscribeReqtoByteArray
 SubscribeReqbyte

SubscribeReqProto.SubscribeReqparseFrom
 byte

SubReqServer

8-2 ProtobufSubReqServer

[illegible]

36. `ch.pipeline().addLast(`

37. `new ProtobufDecoder(`

38. `SubscribeReqProto.SubscribeReq`

39. `.getDefaultInstance()));`

```
40. ch.pipeline().addLast(
```

```
41.                                     new  
ProtobufVarint32LengthFieldPrepender());
```

```
42.                                     ch.pipeline().addLast(new  
ProtobufEncoder());
```

```
43.                                     ch.pipeline().addLast(new  
SubReqServerHandler());
```

```

44.         }
45.         });
46.
47.         // 關閉通道
48.         ChannelFuture f = b.bind(port).sync();
49.
50.         // 關閉通道
51.         f.channel().closeFuture().sync();
52.     } finally {
53.         // 關閉通道
54.         bossGroup.shutdownGracefully();
55.         workerGroup.shutdownGracefully();
56.     }
57. }
58.
59.     public static void main(String[] args) throws
Exception {
60.         int port = 8080;
61.         if (args != null && args.length > 0) {
62.             try {
63.                 port = Integer.valueOf(args[0]);
64.             } catch (NumberFormatException e) {
65.                 // 關閉通道
66.             }
67.         }

```

```

68.    new SubReqServer().bind(port);
69.    }
70. }

```

```

    34    ①    ②    ③    ④    ChannelPipeline    ⑤    ⑥
ProtobufVarint32FrameDecoder    ⑦    ⑧    ⑨    ⑩    ⑪    ⑫    ⑬    ⑭
ProtobufDecoder    ⑮    ⑯    ⑰    ⑱    ⑲    ⑳    ㉑    ㉒
com.google.protobuf.MessageLite    ㉓    ㉔    ㉕    ㉖    ㉗    ㉘    ㉙
ProtobufDecoder    ㉚    ㉛    ㉜    ㉝    ㉞    ㉟    ㊱    ㊲    ㊳    ㊴    ㊵    ㊶
    ㊷    ㊸    ㊹

```

```

    ㊸    ㊹    ㊺    SubReqServerHandler    ㊻    ㊼

```

```

    ㊸    ㊹    ㊺    8-3    Protobuf    ㊻    ㊼    SubReqServerHandler

```

```

11. @Sharable
    12.    public    class    SubReqServerHandler    extends
ChannelHandlerAdapter {
13.
14.    @Override
15.        public void channelRead(ChannelHandlerContext ctx,
Object msg)
16.            throws Exception {
17.                SubscribeReqProto.SubscribeReq    req    =
(SubscribeReqProto.SubscribeReq) msg;
18.        if ("Lilinfeng".equalsIgnoreCase(req.getUserName()))
{

```

```

19.          System.out.println("Service accept client
subscribe req : ["
20.          + req.toString() + "]);
21.          ctx.writeAndFlush(resp(req.getSubReqID()));
22.      }
23.  }
24.
25.      private SubscribeRespProto.SubscribeResp resp(int
subReqID) {
26.          SubscribeRespProto.SubscribeResp.Builder builder =
SubscribeRespProto.SubscribeResp
27.          .newBuilder();
28.          builder.setSubReqID(subReqID);
29.          builder.setRespCode(0);
30.          builder.setDesc("Netty book order succeed, 3 days
later, sent to the designated address");
31.          return builder.build();
32.      }
33.
34.      @Override
35.          public void exceptionCaught(ChannelHandlerContext
ctx, Throwable cause) {
36.              cause.printStackTrace();
37.              ctx.close();// 关闭连接
38.          }
39.  }

```

ProtobufDecoder和ProtobufEncoder
SubscribeRespProto.SubscribeResp

8.2.2 Protobuf

demo Protobuf

8-4 Protobuf SubReqClient

```
20. public class SubReqClient {
21.
22.     public void connect(int port, String host) throws
Exception {
23.         // NIO
24.         EventLoopGroup group = new NioEventLoopGroup();
25.         try {
26.             Bootstrap b = new Bootstrap();
27.             b.group(group).channel(NioSocketChannel.class)
28.                 .option(ChannelOption.TCP_NODELAY, true)
29.                 .handler(new
ChannelInitializer<SocketChannel>() {
30.                     @Override
31.                     public void initChannel(SocketChannel ch)
32.                         throws Exception {
33.                         ch.pipeline().addLast(
```

34. `new ProtobufVarint32FrameDecoder());`

35. `ch.pipeline().addLast(`

36. `new ProtobufDecoder(`

37. `SubscribeRespProto.SubscribeResp`

38. `.getDefaultInstance());`

39. `ch.pipeline().addLast(`

40. `new
ProtobufVarint32LengthFieldPrepender());`

41. `ch.pipeline().addLast(new
ProtobufEncoder());`

```
42.                                     ch.pipeline().addLast(new
SubReqClientHandler());
```

```
43.                                     }
44.                                     });
45.
46.                                     // 连接超时
47.                                     ChannelFuture f = b.connect(host, port).sync();
48.
49.                                     // 关闭超时
50.                                     f.channel().closeFuture().sync();
51.     } finally {
52.         // 关闭NIO 线程
53.         group.shutdownGracefully();
54.     }
55. }
56.
57. /**
58.  * @param args
59.  * @throws Exception
60.  */
61.     public static void main(String[] args) throws
```

```

Exception {
    62.     int port = 8080;
    63.     if (args != null && args.length > 0) {
    64.         try {
    65.             port = Integer.valueOf(args[0]);
    66.         } catch (NumberFormatException e) {
    67.             // ignore
    68.         }
    69.     }
    70.     new SubReqClient().connect(port, "127.0.0.1");
    71. }
    72. }

```

37 38 SubscribeResp
 Proto.SubscribeResp

8-5 Protobuf SubReqClientHandler

```

    13.     public class SubReqClientHandler extends
ChannelHandlerAdapter {
    14.
    15.         /**
    16.          * Creates a client-side handler.
    17.          */
    18.         public SubReqClientHandler() {
    19.         }

```

```
20.
21.     @Override
22.     public void channelActive(ChannelHandlerContext ctx)
{
23.         for (int i = 0; i < 10; i++) {
24.             ctx.write(subReq(i));
25.         }
26.         ctx.flush();
27.     }
28.
29.     private SubscribeReqProto.SubscribeReq subReq(int i)
{
30.         SubscribeReqProto.SubscribeReq.Builder builder =
SubscribeReqProto.SubscribeReq
31.             .newBuilder();
32.         builder.setSubReqID(i);
33.         builder.setUserName("Lilinfeng");
34.         builder.setProductName("Netty Book For Protobuf");
35.         List<String> address = new ArrayList<>();
36.         address.add("NanJing YuHuaTai");
37.         address.add("BeiJing LiuLiChang");
38.         address.add("ShenZhen HongShuLin");
39.         builder.addAllAddress(address);
40.         return builder.build();
41.     }
42.
43.     @Override
```

```

44.      public void channelRead(ChannelHandlerContext ctx,
Object msg)
45.          throws Exception {
46.      System.out.println("Receive server response : [" +
msg + "]);
47.  }
48.
49.  @Override
50.      public void
channelReadComplete(ChannelHandlerContext ctx) throws Exception
{
51.      ctx.flush();
52.  }
53.
54.  @Override
55.      public void exceptionCaught(ChannelHandlerContext
ctx, Throwable cause) {
56.      cause.printStackTrace();
57.      ctx.close();
58.  }
59. }

```

10

Protobuf

8.2.3 Protobuf

□□□□□□□□□□□□□□□□

□□□□□□□□□□

Service accept client subscribe req : [subReqID: 0
userName: "Lilinfeng"
productName: "Netty Book For Protobuf"
address: "NanJing YuHuaTai"
address: "BeiJing LiuLiChang"
address: "ShenZhen HongShuLin"
]

Service accept client subscribe req : [subReqID: 1
userName: "Lilinfeng"
productName: "Netty Book For Protobuf"
address: "NanJing YuHuaTai"
address: "BeiJing LiuLiChang"
address: "ShenZhen HongShuLin"
]

Service accept client subscribe req : [subReqID: 2
userName: "Lilinfeng"
productName: "Netty Book For Protobuf"
address: "NanJing YuHuaTai"
address: "BeiJing LiuLiChang"
address: "ShenZhen HongShuLin"
]

Service accept client subscribe req : [subReqID: 3
userName: "Lilinfeng"

productName: "Netty Book For Protobuf"
address: "NanJing YuHuaTai"
address: "BeiJing LiuLiChang"
address: "ShenZhen HongShuLin"
]

Service accept client subscribe req : [subReqID: 4
userName: "Lilinfeng"
productName: "Netty Book For Protobuf"
address: "NanJing YuHuaTai"
address: "BeiJing LiuLiChang"
address: "ShenZhen HongShuLin"
]

Service accept client subscribe req : [subReqID: 5
userName: "Lilinfeng"
productName: "Netty Book For Protobuf"
address: "NanJing YuHuaTai"
address: "BeiJing LiuLiChang"
address: "ShenZhen HongShuLin"
]

Service accept client subscribe req : [subReqID: 6
userName: "Lilinfeng"
productName: "Netty Book For Protobuf"
address: "NanJing YuHuaTai"
address: "BeiJing LiuLiChang"
address: "ShenZhen HongShuLin"
]

Service accept client subscribe req : [subReqID: 7

```
userName: "Lilinfeng"  
productName: "Netty Book For Protobuf"  
address: "NanJing YuHuaTai"  
address: "BeiJing LiuLiChang"  
address: "ShenZhen HongShuLin"  
]
```

Service accept client subscribe req : [subReqID: 8

```
userName: "Lilinfeng"  
productName: "Netty Book For Protobuf"  
address: "NanJing YuHuaTai"  
address: "BeiJing LiuLiChang"  
address: "ShenZhen HongShuLin"  
]
```

Service accept client subscribe req : [subReqID: 9

```
userName: "Lilinfeng"  
productName: "Netty Book For Protobuf"  
address: "NanJing YuHuaTai"  
address: "BeiJing LiuLiChang"  
address: "ShenZhen HongShuLin"  
]
```

□□□□□□□□

Receive server response : [subReqID: 0

respCode: 0

desc: "Netty book order succeed, 3 days later, sent to the

designated address"

]

Receive server response : [subReqID: 1

respCode: 0

desc: "Netty book order succeed, 3 days later, sent to the
designated address"

]

Receive server response : [subReqID: 2

respCode: 0

desc: "Netty book order succeed, 3 days later, sent to the
designated address"

]

Receive server response : [subReqID: 3

respCode: 0

desc: "Netty book order succeed, 3 days later, sent to the
designated address"

]

Receive server response : [subReqID: 4

respCode: 0

desc: "Netty book order succeed, 3 days later, sent to the
designated address"

]

Receive server response : [subReqID: 5

respCode: 0

desc: "Netty book order succeed, 3 days later, sent to the
designated address"

]

```

Receive server response : [subReqID: 6
respCode: 0
desc: "Netty book order succeed, 3 days later, sent to the
designated address"
]
Receive server response : [subReqID: 7
respCode: 0
desc: "Netty book order succeed, 3 days later, sent to the
designated address"
]
Receive server response : [subReqID: 8
respCode: 0
desc: "Netty book order succeed, 3 days later, sent to the
designated address"
]
Receive server response : [subReqID: 9
respCode: 0
desc: "Netty book order succeed, 3 days later, sent to the
designated address"
]

```

Netty Protobuf
 Netty Protobuf Protobuf
 Protobuf

8.3 Protobuf

ProtobufDecoder 和 ProtobufDecoder 的构造函数如下：

- Netty 的 ProtobufVarint32FrameDecoder 构造函数如下：
- Netty 的 LengthFieldBasedFrameDecoder 构造函数如下：
- ByteToMessageDecoder 构造函数如下：

ProtobufDecoder 的构造函数如下：

ProtobufVarint32FrameDecoder 的构造函数如下：

图 8-8 ProtobufVarint32FrameDecoder 的构造函数

图 8-9 ProtobufVarint32FrameDecoder 的构造函数

图 8-9 ProtobufVarint32FrameDecoder 的构造函数

8.4 小结

Protobuf 是一种轻量级的数据交换格式。Protobuf 的 POJO 类与 Protobuf 的 Netty 的 Protobuf 类。Protobuf 的 Protobuf 类。

——JBoss 的 Marshalling 和 Netty 的 Marshalling。JBoss 的 Marshalling 和 Netty 的 Marshalling。JBoss 的 Marshalling 和 Netty 的 Marshalling。

第9章 JBoss Marshalling

JBoss Marshalling 是 Java 序列化 API 的扩展，它允许你使用 `java.io.Serializable` 接口来序列化对象，而不需要实现 `Serializable` 接口。

JBoss Marshalling 的优点：

- Marshalling 接口简单
- Netty 与 Marshalling 集成
- Netty 与 Marshalling 集成
- 支持 Marshalling

9.1 Marshalling 接口

JBoss Marshalling 接口位于 `org.jboss.marshalling` 包中。你可以从 `jboss-marshalling-1.3.0` 或 `jboss-marshalling-serial-1.3.0` 中下载。

<https://www.jboss.org/jbossmarshalling/downloads>

JBoss Marshalling API 与 JBoss Marshalling Serial Protocol 接口在 9-1 中。

图 9-1 Marshalling 接口

JBoss Marshalling API 的类路径在 9-2 中。

图 9-2 JBoss Marshalling 类路径

Marshalling Marshalling

9.2 Netty的Marshalling和Unmarshalling

POJOJBossMarshallingJDK
7SubscribeReqSubscribeRespJBossAPI
SubscribeReqSubscribeRespNettyJBoss
NettyMarshalling

□ □ □ □ □ □ □

[illegible]

9-1 Marshalling SubReqServer

```

18.     public class SubReqServer {
19.         public void bind(int port) throws Exception {
20.             // 创建NIO线程
21.             EventLoopGroup bossGroup = new
NioEventLoopGroup();
22.             EventLoopGroup workerGroup = new
NioEventLoopGroup();
23.             try {
24.                 ServerBootstrap b = new ServerBootstrap();
25.                 b.group(bossGroup, workerGroup)
26.                     .channel(NioServerSocketChannel.class)

```



```
27.                .option(ChannelOption.SO_BACKLOG, 100)
28.                .handler(new
LoggingHandler(LogLevel.INFO))
29.                .childHandler(new
ChannelInitializer<SocketChannel>() {
30.                    @Override
31.                    public void initChannel(SocketChannel ch)
{
32.                        ch.pipeline().addLast(
```

```
33.                MarshallingCodeCFactory
```

```
34.                .buildMarshallingDecoder());
```

```
35.                ch.pipeline().addLast(
```

36. *MarshallingCodeCFactory*

37. *.buildMarshallingEncoder());*

38. *ch.pipeline().addLast(new
SubReqServerHandler());*

39. *}*

40. *});*

41.

42. *// 启动服务器*

43. *ChannelFuture f = b.bind(port).sync();*

44.

45. *// 等待服务器启动*

46. *f.channel().closeFuture().sync();*

```

47.         } finally {
48.             // 关闭线程池
49.             bossGroup.shutdownGracefully();
50.             workerGroup.shutdownGracefully();
51.         }
52.     }
53.
54.     public static void main(String[] args) throws
Exception {
55.         int port = 8080;
56.         if (args != null && args.length > 0) {
57.             try {
58.                 port = Integer.valueOf(args[0]);
59.             } catch (NumberFormatException e) {
60.                 // 打印异常
61.             }
62.         }
63.         new SubReqServer().bind(port);
64.     }
65. }

```

32 34 36 38 40 42 44 46 48 50 52 54 56 58 60 62 64 66 68 70 72 74 76 78 80 82 84 86 88 90 92 94 96 98 100
 MarshallingDecoder 35 37 39 41 43 45 47 49 51 53 55 57 59 61 63 65 67 69 71 73 75 77 79 81 83 85 87 89 91 93 95 97 99 101
 MarshallingEncoder 35 37 39 41 43 45 47 49 51 53 55 57 59 61 63 65 67 69 71 73 75 77 79 81 83 85 87 89 91 93 95 97 99 101

MarshallingCodeCFactory 35 37 39 41 43 45 47 49 51 53 55 57 59 61 63 65 67 69 71 73 75 77 79 81 83 85 87 89 91 93 95 97 99 101

□ □ □ □ 9-2 Marshalling □ □ □ □ □ □ □ MarshallingCodeCFactory

```
18.  public final class MarshallingCodeCFactory {
19.
20.      /**
21.       * □□Jboss Marshalling□□□MarshallingDecoder
22.       *
23.       * @return
24.       */
25.          public static MarshallingDecoder
buildMarshallingDecoder() {
26.          final MarshallerFactory marshallerFactory =
Marshalling

27.          .getProvidedMarshallerFactory("serial");

28.          final MarshallingConfiguration configuration =
```

```
new MarshallingConfiguration();
```

```
29. configuration.setVersion(5);
```

```
30. UnmarshallerProvider provider = new  
DefaultUnmarshallerProvider(
```

```
31. marshallerFactory, configuration);
```

```
32. MarshallingDecoder decoder = new  
MarshallingDecoder(provider, 1024);
```

```
33.         return decoder;
```

```
34.     }
```

```
35.
```

```
36.     /**
```

```
37.      *  Jboss Marshalling MarshallingEncoder
```

```
38.      *
```

```
39.      * @return
```

```
40.      */
```

```
41.         public static MarshallingEncoder  
buildMarshallingEncoder() {
```

```
42.         final MarshallerFactory marshallerFactory =  
Marshalling
```

43. `.getProvidedMarshallerFactory("serial");`

44. `final MarshallingConfiguration configuration =
new MarshallingConfiguration();`

45. `configuration.setVersion(5);`

46. `MarshallerProvider provider = new
DefaultMarshallerProvider(`

47. `marshallerFactory, configuration);`

```
48.         MarshallingEncoder encoder = new  
MarshallingEncoder(provider);
```

```
49.         return encoder;
```

```
50.     }
```

```
51. }
```

在 26 和 27 行，我们使用 Marshalling 包中的 `getProvidedMarshallerFactory` 方法，该方法返回一个 `MarshallerFactory` 对象，该对象用于序列化 Java 对象。我们使用 `jboss-marshalling-serial-1.3.0.CR9.jar` 包。

第28行代码MarshallConfiguration的构造函数中5行代码
MarshallFactory 从 MarshallConfiguration 中 从
UnmarshallProvider 中 从 从 从 从 从 从 从 从 从 从 Netty 中
MarshallDecoder的构造函数中UnmarshallProvider的
构造函数中

第42行代码45行代码 MarshallFactory 从
MarshallConfiguration中46行代码MarshallProvider的
构造函数中Netty的MarshallEncoder的MarshallEncoder的
构造函数中POJO的构造函数中

第 SubReqServerHandler 的 8 行代码
SubReqServerHandler的构造函数中

9.3 Netty的Marshall

在TCP的/的10行代码
ChannelPipeline的MarshallEncoder的
POJO的Marshall的
MarshallDecoder的

图9-3 Marshall的 SubReqClient

```

16.    public class SubReqClient {
17.
18.        public void connect(int port, String host) throws
Exception {
19.            // 创建NIO
20.            EventLoopGroup group = new NioEventLoopGroup();
21.            try {
22.                Bootstrap b = new Bootstrap();
23.
b.group(group).channel(NioSocketChannel.class)
24.                .option(ChannelOption.TCP_NODELAY, true)
25.                .handler(new
ChannelInitializer<SocketChannel>() {
26.                    @Override
27.                    public void initChannel(SocketChannel ch)
28.                        throws Exception {
29.                        ch.pipeline().addLast(

```

30. *MarshallngCodeCFactory*

31. `.buildMarshallingDecoder());`

32. `ch.pipeline().addLast(`

33. `MarshallingCodeCFactory`

34. `.buildMarshallingEncoder());`

35. `ch.pipeline().addLast(new
SubReqClientHandler());`

```

36.         }
37.         });
38.
39.         // 關閉通道
40.         ChannelFuture f = b.connect(host,
port).sync();
41.
42.         // 關閉通道
43.         f.channel().closeFuture().sync();
44.     } finally {
45.         // 關閉NIO選擇器
46.         group.shutdownGracefully();
47.     }
48. }
49.
50. /**
51.  * @param args
52.  * @throws Exception
53.  */
54.     public static void main(String[] args) throws
Exception {
55.         int port = 8080;
56.         if (args != null && args.length > 0) {
57.             try {
58.                 port = Integer.valueOf(args[0]);
59.             } catch (NumberFormatException e) {
60.                 // 關閉通道

```

```

61.         }
62.     }
63.     new SubReqClient().connect(port, "127.0.0.1");
64. }
65. }

```

图 29 图 34 中 8 个 Marshalling 对象通过 ChannelPipeline 连接

SubReqClientHandler 中 8 个 Marshalling 对象通过 ChannelPipeline 连接

图 9-3 中 8 个 Marshalling 对象

9.4 图 9-3 Marshalling 对象

图 9-3 中 8 个 Marshalling 对象

图 9-3 中 8 个 Marshalling 对象

```

Service accept client subscrib req : [SubscribeReq
[subReqID=0, userName= Lilinfeng, productName=Netty Book For
Marshalling,      phoneNumber=138xxxxxxxx,      address=NanJing
YuHuaTai]]

```

```

Service accept client subscrib req : [SubscribeReq
[subReqID=1, userName= Lilinfeng, productName=Netty Book For
Marshalling,      phoneNumber=138xxxxxxxx,      address=NanJing
YuHuaTai]]

```

Service accept client subscrib req : [SubscribeReq
[subReqID=2, userName= Lilinfeng, productName=Netty Book For
Marshalling, phoneNumber=138xxxxxxxxx, address=NanJing
YuHuaTai]]

Service accept client subscrib req : [SubscribeReq
[subReqID=3, userName= Lilinfeng, productName=Netty Book For
Marshalling, phoneNumber=138xxxxxxxxx, address=NanJing
YuHuaTai]]

Service accept client subscrib req : [SubscribeReq
[subReqID=4, userName= Lilinfeng, productName=Netty Book For
Marshalling, phoneNumber=138xxxxxxxxx, address=NanJing
YuHuaTai]]

Service accept client subscrib req : [SubscribeReq
[subReqID=5, userName= Lilinfeng, productName=Netty Book For
Marshalling, phoneNumber=138xxxxxxxxx, address=NanJing
YuHuaTai]]

Service accept client subscrib req : [SubscribeReq
[subReqID=6, userName= Lilinfeng, productName=Netty Book For
Marshalling, phoneNumber=138xxxxxxxxx, address=NanJing
YuHuaTai]]

Service accept client subscrib req : [SubscribeReq
[subReqID=7, userName= Lilinfeng, productName=Netty Book For
Marshalling, phoneNumber=138xxxxxxxxx, address=NanJing
YuHuaTai]]

Service accept client subscrib req : [SubscribeReq
[subReqID=8, userName= Lilinfeng, productName=Netty Book For
Marshalling, phoneNumber=138xxxxxxxxx, address=NanJing

Receive server response : [SubscribeResp [subReqID=5, respCode=0, desc=Netty book order succeed, 3 days later, sent to the designated address]]

Receive server response : [SubscribeResp [subReqID=6, respCode=0, desc=Netty book order succeed, 3 days later, sent to the designated address]]

Receive server response : [SubscribeResp [subReqID=7, respCode=0, desc=Netty book order succeed, 3 days later, sent to the designated address]]

Receive server response : [SubscribeResp [subReqID=8, respCode=0, desc=Netty book order succeed, 3 days later, sent to the designated address]]

Receive server response : [SubscribeResp [subReqID=9, respCode=0, desc=Netty book order succeed, 3 days later, sent to the designated address]]

10個のsubReqID(0~9)のSubscribeRespを受け取る

TCP/IPの通信をNettyのMarshallingとChannelPipelineのMarshallingで処理する

NettyのMarshallingはBossのThreadで処理される

9.5 通信

Netty Marshalling POJO
Netty Marshalling JBoss Marshalling
JBoss Jboss
Marshalling

Netty HTTP HTTP
HTTP Netty HTTP
Netty HTTP

Netty

10 HTTP

11 WebSocket

12 UDP

13

14

第10章 HTTP 协议

HTTP 协议是建立在 TCP 协议基础上的，是 Internet 上应用最广泛、最重要的协议。HTTP 协议是由 IETF 组织在 1990 年提出的，旨在为 Web 浏览器和 Web 服务器之间提供一种简单的、无状态的、基于文本的通信协议。

HTTP 协议是 Web 应用的核心，也是 HTTP 应用的基础。Netty 是一个高性能的 HTTP 客户端和服务器框架，它支持 HTTP 1.0 和 HTTP 1.1 协议。Netty 的 HTTP 模块是基于 Netty 的 NIO 模块实现的，它提供了对 HTTP 协议的完整支持。

本章主要介绍：

- HTTP 协议
- Netty HTTP 客户端
- HTTP + XML
- HTTP 服务器

10.1 HTTP 概述

HTTP 协议是建立在 TCP 协议基础上的，是 Internet 上应用最广泛、最重要的协议。

HTTP 协议的主要特点：

- 基于 Client/Server 模式
- 无状态——每次请求都是独立的，不依赖于之前的请求。URL 是唯一的标识符。
- 简单——HTTP 协议使用简单的文本格式，易于理解和实现。HTTP 响应头包含 Content-Type 等信息。

- 通常——HTTP 客户端和服务端通过 TCP 连接进行通信，连接建立后，客户端发送请求，服务端返回响应。

10.1.1 HTTP 的 URL

HTTP URL 的格式如下：URL 由 scheme、host、port、abs_path 组成。

```
http://host[":"port][abs_path]
```

其中，http 是 HTTP 协议的名称，host 是 Internet 上的主机 IP 地址，port 是端口号，默认为 80，abs_path 是绝对 URI 路径，URL 的绝对路径由 abs_path 和 URI 组成，"/" 表示根路径。

10.1.2 HTTP 的 Http Request

HTTP 请求的格式如下：

- HTTP 请求行
- HTTP 请求头
- HTTP 请求体

请求行的格式如下：URI Method Request-URI HTTP-Version CRLF

MethodRequest-URIHTTP-Version
HTTPCRLFCRLFCRLF
LF

- GETRequest-URI
- POSTRequest-URI
- HEADRequest-URI
- PUTRequest-URI
- DELETERequest-URI
- TRACE
- CONNECT
- OPTIONS

GETGET
http://localhost:8080/netty-5.0.010-1

10-1 Netty HTTP

HTTP

```
GET /netty5.0 HTTP/1.1
Host: localhost:8080
Connection: keep-alive
User-Agent: Mozilla/5.0 (Windows NT 5.1) AppleWebKit/537.1
(KHTML, like Gecko) Chrome/21.0.1180.89 Safari/537.1
```

Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Encoding: gzip,deflate,sdch
Accept-Language: zh-CN,zh;q=0.8
Accept-Charset: GBK,utf-8;q=0.7,*;q=0.3
Content-Length: 0

GET

POST
GET
POST

1 HTTP GET POST

2 GET URL request line
" " URL "&" POST HTTP

3 URL IE URL
2083 2K+35 GET POST
URL

4 POST GET GET
URL 1 2
GET Cross-site request
forgery POST

10-1
10-1

10-1 HTTP

HTTP+XML
XML

10.1.3 HTTP

HTTP
HTTP

HTTP-Version Status-Code Reason-Phrase
CRLF HTTP-Version HTTP Status-Code
Status-Code

5

1 1xx

2 2xx

3 3xx

4 4xx

5 5xx

10-2

图10-2 HTTP请求报文格式

请求报文格式如下所示，其中Request-URI为请求资源的路径，如图10-3所示。

图10-3 请求报文

10.2 Netty HTTP客户端

Netty的HTTP客户端使用HTTP客户端接口，Netty的HTTP客户端使用NIO TCP客户端实现HTTP客户端。

Netty的HTTP客户端使用Web客户端接口，Tomcat、Jetty、Web客户端接口。

10.2.1 HTTP客户端

Netty的HTTP客户端使用HTTP客户端接口，HTTP客户端接口使用HTTP 403客户端接口。

Netty的HTTP客户端使用HTTP客户端接口，HTTP客户端接口使用HTTP客户端接口。

Netty的HTTP客户端使用HTTP客户端接口。

10.2.2 HTTP客户端

HTTP客户端接口。

例10-1 HTTPサーバー 例 HttpFileServer

```
19.  public class HttpFileServer {
20.
21.      private static final String DEFAULT_URL =
"/src/com/phei/netty/";
22.
23.      public void run(final int port, final String url)
throws Exception {
24.          EventLoopGroup bossGroup = new
NioEventLoopGroup();
25.          EventLoopGroup workerGroup = new
NioEventLoopGroup();
26.      try {
27.          ServerBootstrap b = new ServerBootstrap();
28.          b.group(bossGroup, workerGroup)
29.            .channel(NioServerSocketChannel.class)
30.            .childHandler(new
ChannelInitializer<SocketChannel>() {
31.                @Override
32.                protected void initChannel(SocketChannel
ch)
33.                    throws Exception {
34.                    ch.pipeline().addLast("http-decoder",
```

35. `new HttpRequestDecoder());`

36. `ch.pipeline().addLast("http-
aggregator",`

37. `new HttpObjectAggregator(65536));`

38. `ch.pipeline().addLast("http-encoder",`

39. `new HttpResponseEncoder());`

40. `ch.pipeline().addLast("http-chunked",`

41. `new ChunkedWriteHandler());`

42. `ch.pipeline().addLast("fileServerHandler",`

43. `new HttpFileServerHandler(url));`

```

44.         }
45.     });

46.         ChannelFuture future =
b.bind("192.168.1.102", port).sync();
47.         System.out.println("HTTP服务器地址 : " +
"http://192.168.1.102:"
48.         + port + url);
49.         future.channel().closeFuture().sync();
50.     } finally {
51.         bossGroup.shutdownGracefully();
52.         workerGroup.shutdownGracefully();
53.     }
54. }
55.
56.     public static void main(String[] args) throws
Exception {
57.         int port = 8080;
58.         if (args.length > 0) {
59.             try {
60.                 port = Integer.parseInt(args[0]);
61.             } catch (NumberFormatException e) {
62.                 e.printStackTrace();

```

```

63.         }
64.     }
65.     String url = DEFAULT_URL;
66.     if (args.length > 1)
67.         url = args[1];
68.     new HttpFileServer().run(port, url);
69. }
70. }

```

在main方法中，我们指定了HTTP的URL，并指定了端口号为8080，URL为“/src/com/phei/netty/”

在34-43行，我们定义了ChannelPipeline，并指定了HTTP的Handler，包括HttpObjectAggregator，FullHttpRequest，FullHttpResponse，以及HTTP的Handler。

1. HttpRequest / HttpResponse

2. HttpContent

3. LastHttpContent

38-39行，我们指定了HTTP的Handler，包括40-41行的Chunked handler，以及Java的Handler。

□□□□HttpFileServerHandler□□□□□□□□□□□□□□□□□□□□
□□□□□□□□

□□□□10-2 HTTP□□□□□ □□□HttpFileServerHandler

```
47.    public class HttpFileServerHandler extends
48.        SimpleChannelInboundHandler<FullHttpRequest> {
49.        private final String url;
50.
51.        public HttpFileServerHandler(String url) {
52.            this.url = url;
53.        }
54.
55.        @Override
56.        public void messageReceived(ChannelHandlerContext
ctx,
57.            FullHttpRequest request) throws Exception {
58.            if (!request.getDecoderResult().isSuccess()) {
59.                sendError(ctx, BAD_REQUEST);
60.                return;
61.            }
62.            if (request.getMethod() != GET) {
63.                sendError(ctx, METHOD_NOT_ALLOWED);
64.                return;
65.            }
66.            final String uri = request.getUri();
67.            final String path = sanitizeUri(uri);
```



```

68.         if (path == null) {
69.             sendError(ctx, FORBIDDEN);
70.             return;
71.         }
72.         File file = new File(path);
73.         if (file.isHidden() || !file.exists()) {
74.             sendError(ctx, NOT_FOUND);
75.             return;
76.         }
77.         if (file.isDirectory()) {
78.             if (uri.endsWith("/")) {
79.                 sendListing(ctx, file);
80.             } else {
81.                 sendRedirect(ctx, uri + '/');
82.             }
83.             return;
84.         }
85.         if (!file.isFile()) {
86.             sendError(ctx, FORBIDDEN);
87.             return;
88.         }
89.         RandomAccessFile randomAccessFile = null;
90.         try {
91.             randomAccessFile = new RandomAccessFile(file,
"r");// 读取文件
92.         } catch (FileNotFoundException fnfe) {
93.             sendError(ctx, NOT_FOUND);

```

```

94.            return;
95.        }
96.        long fileLength = randomAccessFile.length();
97.            HttpResponse response = new
DefaultHttpResponse(HTTP_1_1, OK);
98.        setContentLength(response, fileLength);
99.        setContentTypeHeader(response, file);
100.        if (isKeepAlive(request)) {
101.            response.headers().set(CONNECTION,
HttpHeaders.Values.KEEP_ALIVE);
102.        }
103.        ctx.write(response);
104.        ChannelFuture sendFileFuture;
105.            sendFileFuture=ctx.write(new
ChunkedFile(randomAccessFile, 0,
106.            fileLength, 8192),
ctx.newProgressivePromise());
107.            sendFileFuture.addListener(new
ChannelProgressiveFutureListener() {
108.                @Override
109.                    public void
operationProgressed(ChannelProgressiveFuture future,
110.                    long progress, long total) {
111.                    if (total < 0) { // total unknown
112.                        System.err.println("Transfer
progress: " + progress);
113.                    } else {

```

```

114.                System.err.println("Transfer
progress:"+progress+"/"
115.                + total);
116.            }
117.        }
118.
119.        @Override
120.            public void
operationComplete(ChannelProgressiveFuture future)
121.                throws Exception {
122.                System.out.println("Transfer complete.");
123.            }
124.        });
125.        ChannelFuture lastContentFuture = ctx
126.
.writeAndFlush(LastHttpContent.EMPTY_LAST_CONTENT);
127.        if (!isKeepAlive(request)) {
128.
lastContentFuture.addListener(ChannelFutureListener.CLOSE);
129.        }
130.    }
131.
132.    @Override
133.        public void
exceptionCaught(ChannelHandlerContext ctx, Throwable cause)
134.            throws Exception {
135.            cause.printStackTrace();

```

```
136.         if (ctx.channel().isActive()) {
137.             sendError(ctx, INTERNAL_SERVER_ERROR);
138.         }
139.     }
140.
141.         private static final Pattern INSECURE_URI =
Pattern.compile (".*[<>&\\"].*");
142.
143.         private String sanitizeUri(String uri) {
144.             try {
145.                 uri = URLDecoder.decode(uri, "UTF-8");
146.             } catch (UnsupportedEncodingException e) {
147.                 try {
148.                     uri = URLDecoder.decode(uri, "ISO-8859-
149.                 } catch (UnsupportedEncodingException e1)
150.             {
151.                 throw new Error();
152.             }
153.             if (!uri.startsWith(url)) {
154.                 return null;
155.             }
156.             if (!uri.startsWith("/")) {
157.                 return null;
158.             }
159.             uri = uri.replace('/', File.separatorChar);
```

```

160.         if (uri.contains(File.separator + '.'))
161.                                                     ||
uri.contains('.') + File.separator) || uri.startsWith(".")
162.                                                     ||
uri.endsWith(".") || INSECURE_URI.matcher(uri).matches()){
163.             return null;
164.         }
165.             return System.getProperty("user.dir") +
File.separator + uri;
166.         }
167.
168.             private static final Pattern
ALLOWED_FILE_NAME = Pattern
169.                 .compile("[A-Za-z0-9][_A-Za-z0-9\\.]*");
170.
171.             private static void
sendListing(ChannelHandlerContext ctx, File dir) {
172.                 FullHttpResponse response = new
DefaultFullHttpResponse (HTTP_1_1, OK);
173.
response.headers().set(CONTENT_TYPE, "text/html; charset=UTF-8");
174.         StringBuilder buf = new StringBuilder();
175.         String dirPath = dir.getPath();
176.         buf.append("<!DOCTYPE html>\r\n");
177.         buf.append("<html><head><title>");
178.         buf.append(dirPath);
179.         buf.append("  ");

```

```

180.         buf.append("</title></head><body>\r\n");
181.         buf.append("<h3>");
182.         buf.append(dirPath).append("  ");
183.         buf.append("</h3>\r\n");
184.         buf.append("<ul>");
185.             buf.append("<li>   <a href=\"../\">../</a>
</li>\r\n");</li>
186.         for (File f : dir.listFiles()) {
187.             if (f.isHidden() || !f.canRead()) {
188.                 continue;
189.             }
190.             String name = f.getName();
191.                                     if
(!ALLOWED_FILE_NAME.matcher(name).matches()) {
192.                 continue;
193.             }
194.             buf.append("<li>   <a href=\"");</li>
195.             buf.append(name);
196.             buf.append("\>");
197.             buf.append(name);
198.             buf.append("</a></li>\r\n");
199.         }
200.         buf.append("</ul></body></html>\r\n");
201.         ByteBuffer buffer = Unpooled.copiedBuffer(buf,
CharsetUtil.UTF_8);
202.         response.content().writeBytes(buffer);
203.         buffer.release();

```

```

204.                ctx.writeAndFlush(response).addListener
(ChannelFutureListener.CLOSE);
205.            }
206.
207.                private static void
sendRedirect(ChannelHandlerContext ctx, String newUri) {
208.                    FullHttpResponse response = new
DefaultFullHttpResponse (HTTP_1_1, FOUND);
209.                    response.headers().set(LOCATION, newUri);
210.                    ctx.writeAndFlush(response).addListener
(ChannelFutureListener.CLOSE);
211.                }
212.
213.                private static void
sendError(ChannelHandlerContext ctx,
214.                    HttpResponseStatus status) {
215.                    FullHttpResponse response=new
DefaultFullHttpResponse(HTTP_1_1,
216.                        status, Unpooled.copiedBuffer("Failure: "
+ status.toString()
217.                            + "\r\n", CharsetUtil.UTF_8));
218.                    response.headers().set(CONTENT_TYPE,
"text/plain; charset= UTF-8");
219.                    ctx.writeAndFlush(response).addListener
(ChannelFutureListener.CLOSE);
220.                }
221.

```

```

222.                                     private static void
setContentTypeHeader(HttpResponse response, File file) {
223.                                     MimeTypesFileTypeMap mimeTypeMap=new
MimeTypesFileTypeMap();
224.                                     response.headers().set(CONTENT_TYPE,
225.                                     mimeTypeMap.getContentType(file.getPath()));
226.                                     }
227.                                     }

```

58 61 HTTP 400
 62 65 GET POST HTTP
 405

67 URL sanitizeUri 145 JDK
 java.net.URLDecoder URL UTF-8 URI
 159
 161 164 URI

path + URI

6871URIHTTP 40372URIFile7376HTTP 404

172HTTP"text/html; charset=UTF-8"174
HTMLHTML
185 ..186199
201202HTTP
writeAndFlushSocketChannel

85
HTTP 4038595HTTP
404

96HTTPcontent lengthcontent type
Keep-AliveConnectionKeep-Alive103
105 106 Netty ChunkedFile
sendFileFuture GenericFutureListener "Transfer
complete."

chunked LastHttpContent
EMPTY_LAST_CONTENTflush
SocketChannel

Keep-Alive

10.2.3 Netty HTTP

HTTP

10-1

10-1 HTTP

URL

http://192.168.1.102:8080/abcde/get?123

图10-2

图10-2 403

URL 69 HTTP 403

`http://192.168.1.102:8080/src/com/phei/netty/`

图10-3

图10-3 Netty

codec 图10-4

图10-4

protobuf 图10-5

10-5 使用 protobuf

10-6

10-6 使用 protobuf

UTF-8
10-7

10-7 使用 UTF-8

10-8

10-8 使用

10-9

10-9 使用

HTTP

Netty HTTP——HTTP
Netty HTTP

HTTP+XML HTTP+XML Netty
HTTP+XML

10.3 Netty HTTP+XML

HTTP HTTP HTTP
HTTP+XML RESTful+JSON

Java HTTP Servlet Tomcat Web
Jetty Web HTTP HTTP
Web Tomcat Web

Web Tomcat
Web
HTTP

Netty HTTP HTTP+XML

10.3.1 〇〇〇〇〇〇

```

00000000000000000000000000000000HTTP000000000000000000
HTTP00000XML00000HTTP+XML0000000000HTTP000000000000000000
00HTTP+XML0000000000000000HTTP1.100000000CLOSE0000000000HTTP000
000000000000000000000000

```

□□□□□□□□10-3□□□

□10-3 □□□□□□□□Order□

□□□□□□□□10-4□□□

10-4 Customer

□□□□□□□□10-5□□□

□10-5 □□□□□□□Address□

10-6

10-6 Shipping

XML Schema

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:tns="http://phei.com/netty/protocol/http/xml/pojo"
elementFormDefault="qualified"
targetNamespace="http://phei.com/netty/protocol/http/xml/pojo">
  <xs:element type="tns:order" name="order"/>
  <xs:complexType name="address">
    <xs:sequence>
      <xs:element type="xs:string" name="street1"
minOccurs="0"/>
      <xs:element type="xs:string" name="street2"
minOccurs="0"/>
      <xs:element type="xs:string" name="city"
minOccurs="0"/>
      <xs:element type="xs:string" name="state"
minOccurs="0"/>
      <xs:element type="xs:string" name="postCode"
```

```

minOccurs="0"/>
                <xs:element type="xs:string" name="country"
minOccurs="0"/>
            </xs:sequence>
        </xs:complexType>
        <xs:complexType name="order">
            <xs:sequence>
                <xs:element name="customer" minOccurs="0">
                    <xs:complexType>
                        <xs:sequence>
                            <xs:element type="xs:string" name="firstName"
minOccurs="0"/>
                                <xs:element type="xs:string" name="lastName"
minOccurs="0"/>
                                    <xs:element type="xs:string" name="middleName"
minOccurs="0" maxOccurs="unbounded"/>
                                        </xs:sequence>
                                            <xs:attribute
type="xs:long" use="required" name="customerNumber"/>
                                </xs:complexType>
                            </xs:element>
                                <xs:element type="tns:address" name="billTo"
minOccurs="0"/>
                                    <xs:element name="shipping" minOccurs="0">
                                        <xs:simpleType>
                                            <xs:restriction base="xs:string">
                                                <xs:enumeration value="STANDARD_MAIL"/>

```



```

        <xs:enumeration value="PRIORITY_MAIL"/>
        <xs:enumeration value="INTERNATIONAL_MAIL"/>
        <xs:enumeration value="DOMESTIC_EXPRESS"/>
        <xs:enumeration value="INTERNATIONAL_EXPRESS"/>
    </xs:restriction>
</xs:simpleType>
</xs:element>

    <xs:element type="tns:address" name="shipTo"
minOccurs="0"/>
</xs:sequence>

    <xs:attribute type="xs:long" use="required"
name="orderNumber"/>
    <xs:attribute type="xs:float" name="total"/>
</xs:complexType></xs:schema>

```

XML Schema Schema XML XML
W3C XML

HTTP+XML

10.3.2 HTTP+XML

Netty HTTP HTTP+XML 10-10

10-10 HTTP+XML

1 HTTP+XML Netty HTTP POJO HTTP+XML HTTP+XML

2 Netty HTTP HTTP Netty

3 HTTP HTTP+XML POJO HTTP XML Netty POJO Netty

4 XML HTTP Netty HTTP POJO XML

5 HTTP HTTP+XML POJO HTTP Netty

Netty

1 XML POJO - XML
XML

2 HTTP + XML XML - POJO

3 HTTP + XML HTTP

4 HTTP + XML HTTP

5 HTTP + XML HTTP

6 HTTP + XML HTTP

7 HTTP + XML POJO

XML HTTP + XML

10.3.3 XMLとJiBX

JiBXはXML(Extensible Markup Language)とJavaの間の橋渡しをするライブラリです。XMLをJavaのオブジェクトに変換したり、JavaのオブジェクトをXMLに変換したりすることができます。

1 JiBXについて

XMLはExtensible Markup Languageの略で、データを構造的に表現するためのマークアップ言語です。XMLにはDigester、XStream、Castor、JDOM、dom4j、Xalanなどのパースライブラリがあります。JiBXはXMLとJavaの間の橋渡しをするライブラリで、XPathを使用してXMLの特定の要素を取得/設定することができます。

JiBXはXMLとJavaの間の橋渡しをするライブラリで、XMLをJavaのオブジェクトに変換したり、JavaのオブジェクトをXMLに変換したりすることができます。

JiBXはXMLとJavaの間の橋渡しをするライブラリで、XMLをJavaのオブジェクトに変換したり、JavaのオブジェクトをXMLに変換したりすることができます。JiBXはBCEL(Byte Code Engineering Library)とApache Software FoundationのJakartaプロジェクトの一部です。JiBXはJava classworkingとJVMのインターフェースを利用して、XMLとJavaのオブジェクトの間で変換を行います。

JiBx 提供了 Unmarshal 和 Marshal 两个接口。Unmarshal 将 XML 数据转换为 Java 对象，Marshal 将 Java 对象转换为 XML 数据。JiBx 的 Unmarshal/Marshal 接口支持 XPP 和 tree-based 两种模式。DOM (Document Object Model) 和 event stream (SAX, Simple API for Xml) 是 XPP 的两种模式。XML 的解析和生成。

JiBx 提供了 JiBx 和 XML 两个接口。

2 POJO 接口

JiBx 提供了 jar 和 Schema 两个接口。POJO 接口是 JiBx 提供的 Schema 接口 XSD。

POJO 接口是 XML 接口。JiBx 提供了 POJO 接口。Java Bean 接口。

10-3 HTTP+XML POJO Order

```
2 public class Order {
3     private long orderNumber;
```

```

4    private Customer customer;
5
6    /** Billing address information. */
7    private Address billTo;
8
9    private Shipping shipping;
10
11    /**
12     * Shipping address information. If missing, the
billing address is also
13     * used as the shipping address.
14     */
15    private Address shipTo;
16
17    private Float total;
.....//set/get
53}

```

10-4 HTTP+XML POJO Customer

```

2.  import java.util.List;
3.  public class Customer {
4.      private long customerNumber;
5.      /** Personal name. */

```

```

6.      private String firstName;
7.
8.      /** Family name. */
9.      private String lastName;
10.     /** Middle name(s), if any. */
11.     private List<String> middleNames;
.....//set/get
35.     }

```

10-5 HTTP+XML POJO Address

```

2.  public class Address {
3.      /** First line of street information (required).
*/
4.      private String street1;
5.      /** Second line of street information (optional).
*/
6.      private String street2;
7.      private String city;
8.
9.      /**
10.         * State abbreviation (required for the U.S. and
Canada, optional
11.         * otherwise).

```

```

12.         */
13.         private String state;
14.
15.         /** Postal code(required for the U.S.and
Canada,optional otherwise).*/
16.         private String postCode;
17.         /** Country name (optional, U.S. assumed if not
supplied). */
18.         private String country;
.....//set/get
54.     }

```

10-6 HTTP+XML POJO Shipping

```

1.     package com.phei.netty.protocol.http.xml.pojo;
2.
3.     public enum Shipping {
4.         STANDARD_MAIL, PRIORITY_MAIL, INTERNATIONAL_MAIL,
DOMESTIC_EXPRESS, INTERNATIONAL_EXPRESS
5.     }

```

POJO를 생성하는 Ant의 XML, POJO를 생성하는 XML Schema를
생성하는 Ant의

3. Ant의 XML

Eclipse의 Ant window의 Preferences를
Ant 10-11

10-11 Eclipse의 Ant

Eclipse의 Ant를

Ant를
Ant

JiBx task 10-12

10-12 BindGen

JiBx의 org.jibx.binding.generator.BindGen의 POJO Order
Schema 10-13

10-13 AntXML

binding.xmlpojo.xsd10-14

10-14 XML

XMLPOJOXMLOrder10-15

10-15 XMLOrder

JiBxPOJOPOJOAnt10-16

10-16 ClassAnt

10-17

10-17 Class

#####JiBx#####JiBx#####

4 JiBx

JiBx#####

#####10-7 HTTP+XML POJO#####TestOrder

```
18.    public class TestOrder {
19.        private IBindingFactory factory = null;
20.        private StringWriter writer = null;
21.        private StringReader reader = null;
22.        private final static String CHARSET_NAME = "UTF-
8";
23.        private String encode2Xml(Order order) throws
JiBXException, IOException {
24.            factory =
BindingDirectory.getFactory(Order.class);
25.            writer = new StringWriter();
26.            IMarshallingContext mctx =
factory.createMarshallingContext();
27.            mctx.setIndent(2);
28.            mctx.marshalDocument(order, CHARSET_NAME, null,
writer);
```

```

29.      String xmlStr = writer.toString();
30.      writer.close();
31.      System.out.println(xmlStr.toString());
32.      return xmlStr;
33.  }
34.
35.      private Order decode2Order(String xmlBody) throws
JiBXException {
    36.          reader = new StringReader(xmlBody);
        37.              IUnmarshallingContext uctx =
factory.createUnmarshallingContext();
        38.              Order order = (Order)
uctx.unmarshalDocument(reader);
    39.          return order;
    40.      }
    41.
    42.      public static void main(String[] args) throws
JiBXException, IOException {
    43.          TestOrder test = new TestOrder();
    44.          Order order = OrderFactory.create(123);
    45.          String body = test.encode2Xml(order);
    46.          Order order2 = test.decode2Order(body);
    47.          System.out.println(order2);
    48.      }
    49.  }

```

```
24OrderClassIBindingFactory25StringWriter
IBindingFactoryMarshallingmarshalDocumentOrder
StringWriterStringWritertoString()StringXML
```

```
StringReaderStringXML
unmarshalDocumentOrder
```

```
<?xml version="1.0" encoding="UTF-8"?>
<order xmlns="http://phei.com/netty/protocol/http/xml/pojo"
orderNumber= "123" total="9999.999">
  <customer customerNumber="123">
    <firstName></firstName>
    <lastName></lastName>
  </customer>
  <billTo>
    <street1></street1>
    <city></city>
    <state></state>
    <postCode>123321</postCode>
    <country></country>
```

```

</billTo>
<shipping>INTERNATIONAL_MAIL</shipping>
<shipTo>
  <street1>1234</street1>
  <city>ABC</city>
  <state>DEF</state>
  <postCode>123321</postCode>
  <country>GH</country>
</shipTo>
</order>

```

```

Order [orderNumber=123, customer=Customer
[customerNumber=123, firstName= A , lastName= B C ,
middleNames=null], billTo=Address [street1=1234, street2=null,
city= A B C , state= D E F , postCode=123321, country= G H ],
shipping=INTERNATIONAL_MAIL, shipTo=Address [street1= 1 2 3 4 ,
street2=null, city=ABC, state=DEF, postCode=123321, country=G
H], total=9999.999]

```

10.3.3.3 使用XML和JiBx

XML和Netty HTTP +XML

10.3.4 HTTP+XML

第6章 Netty HTTP+XML Netty HTTP

1 HTTP+XML

HTTP+XML

HTTP+XML HTTP Netty HTTP

HTTP+XML ChannelPipeline handler

HTTP+XML

10-8 HTTP+XML HTTP

-
11. public class HttpXmlRequestEncoder extends
 12. AbstractHttpXmlEncoder<HttpXmlRequest> {

```

13.
14.      @Override
15.      protected void encode(ChannelHandlerContext ctx,
HttpXmlRequest msg,
16.          List<Object> out) throws Exception {
17.          ByteBuf body = encode0(ctx, msg.getBody());

18.          FullHttpRequest request = msg.getRequest();

19.          if (request == null) {

20.                                     request = new
DefaultFullHttpRequest(HttpVersion.HTTP_1_1,

```


21. `HttpMethod.GET, "/do", body);`

22. `HttpHeaders headers = request.headers();`

23. `headers.set(HttpHeaders.Names.HOST,
InetAddress.getLocalHost()`

24. `.getHostAddress());`

25. `headers.set(HttpHeaders.Names.CONNECTION,`
`HttpHeaders.Values.CLOSE);`

26. `headers.set(HttpHeaders.Names.ACCEPT_ENCODING,`

27. `HttpHeaders.Values.GZIP.toString() + ', '`

28. `HttpHeaders.Values.DEFLATE.toString());`

29. `headers.set(HttpHeaders.Names.ACCEPT_CHARSET,`

30. `"ISO-8859-1,utf-8;q=0.7,*;q=0.7");`

31. `headers.set(HttpHeaders.Names.ACCEPT_LANGUAGE, "zh");`

32. `headers.set(HttpHeaders.Names.USER_AGENT,`

33. `"Netty xml Http Client side");`

```
34.                headers.set(HttpHeaders.Names.ACCEPT,  
"text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.  
8");
```

```
35.        }
```

```
36.                HttpHeaders.setContentLength(request,  
body.readableBytes());
```

```
37.        out.add(request);
```

38. }

39. }

17 encode0 P0J0Order JiBx XML
Netty ByteBuf 18 HTTP
HTTP

20 35 HTTP HTTP
XML

36 Chunk HTTP Content-
Length XML HTTP out Netty HTTP
HTTP

AbstractHttpXmlEncoder

```
14.      public abstract class AbstractHttpXmlEncoder<T>
extends
15.      MessageToMessageEncoder<T> {
16.      IBindingFactory factory = null;
17.      StringWriter writer = null;
18.      final static String CHARSET_NAME = "UTF-8";
19.      final static Charset UTF_8 =
Charset.forName(CHARSET_NAME);
20.
21.      protected ByteBuf encode0(ChannelHandlerContext
ctx, Object body)
22.      throws Exception {
23.      factory =
BindingDirectory.getFactory(body.getClass());

24.      writer = new StringWriter();
```

```
25.                IMarshallingContext mctx =  
factory.createMarshallingContext();
```

```
26.        mctx.setIndent(2);
```

```
27.        mctx.marshalDocument(body, CHARSET_NAME, null,  
writer);
```

```
28.        String xmlStr = writer.toString();
```

29. *writer.close();*

30. *writer = null;*

31. *ByteBuf encodeBuf = Unpooled.copiedBuffer(xmlStr,*
UTF_8);

32. *return encodeBuf;*

33. *}*


```

34.
35.     @Override
36.     public void exceptionCaught(ChannelHandlerContext
ctx, Throwable cause)
37.     throws Exception {
38.         // 清理
39.         if (writer != null) {
40.             writer.close();
41.             writer = null;
42.         }
43.     }
44. }

```

2330 个字节，JiBx XML 格式，Order 0 个字节，XML 31 个字节，Netty ByteBuf HTTP XML 格式

HttpXmlRequest 格式

10-10 HTTP+XML 格式 HttpXmlRequest

```

9.     public class HttpXmlRequest {
10.         private FullHttpRequest request;

```

```
11.         private Object body;
12.
13.         public HttpRequest(FullHttpRequest request,
Object body) {
14.             this.request = request;
15.             this.body = body;
16.         }
17.
18.         /**
19.          * @return the request
20.          */
21.         public final FullHttpRequest getRequest() {
22.             return request;
23.         }
24.
25.         /**
26.          * @param request
27.          *          the request to set
28.          */
29.         public final void setRequest(FullHttpRequest
request) {
30.             this.request = request;
31.         }
32.
33.         /**
34.          * @return the object
35.          */
```

```

36.      public final Object getBody() {
37.          return body;
38.      }
39.
40.      /**
41.       * @param object
42.       *         the object to set
43.       */
44.      public final void setBody(Object body) {
45.          this.body = body;
46.      }
47.  }

```

FullHttpRequestObject

2 HTTP+XML

HTTP HTTP+XML HTTP JiBx
 POJO handler POJO HTTP

```
20.    public class HttpXmlRequestDecoder extends
21.        AbstractHttpXmlDecoder<FullHttpRequest> {
22.
23.        public HttpXmlRequestDecoder(Class<?> clazz) {
24.            this(clazz, false);
25.        }
26.
27.        public HttpXmlRequestDecoder(Class<?> clazz,
boolean isPrint) {
28.            super(clazz, isPrint);
29.        }
30.
31.        @Override
32.        protected void decode(ChannelHandlerContext arg0,
FullHttpRequest arg1,
33.            List<Object> arg2) throws Exception {
34.            if (!arg1.getDecoderResult().isSuccess()) {

35.                sendError(arg0, BAD_REQUEST);
```

```
36.         return;
```

```
37.     }
```

```
38.         HttpXmlRequest request = new HttpXmlRequest(arg1,  
decode0(arg0,
```

```
39.         arg1.content())));
```

```
40.         arg2.add(request);

41.     }
42.
43.         private static void
sendError(ChannelHandlerContext ctx,
44.         HttpResponseStatus status) {
45.             FullHttpResponse response = new
DefaultFullHttpResponse(HTTP_1_1,

46.             status, Unpooled.copiedBuffer("Failure: " +
status.toString())
```

```
47.                + "\r\n", CharsetUtil.UTF_8));
```

```
48.                response.headers().set(CONTENT_TYPE, "text/plain;  
charset=UTF-8");
```

```
49.                ctx.writeAndFlush(response).addListener(ChannelFutureListener.  
CLOSE);
```

```
50.        }
```

```
51.    }
```

HttpXmlRequestDecoder 是 HTTP 请求的解码器，它实现了 `AbstractHttpXmlDecoder` 接口。它通过 `34` 和 `37` 行代码，从 HTTP 请求中解析出 XML 数据。

`43` 和 `50` 行代码，从 HTTP 请求中解析出 XML 数据。它通过 `43` 行代码，从 HTTP 请求中解析出 XML 数据。

`38` 行代码，从 HTTP 请求中解析出 XML 数据。它通过 `38` 行代码，从 HTTP 请求中解析出 XML 数据。

它通过 `AbstractHttpXmlDecoder` 接口。

10-12 HTTP+XML HTTP 请求的解码器 `AbstractHttpXmlDecoder`

```
18.     public abstract class AbstractHttpXmlDecoder<T>
extends
19.         MessageToMessageDecoder<T> {
20.         private IBindingFactory factory;
21.         private StringReader reader;
22.         private Class<?> clazz;
```



```
23.         private boolean isPrint;
24.         private final static String CHARSET_NAME = "UTF-
8";
25.             private final static Charset UTF_8 =
Charset.forName(CHARSET_NAME);
26.
27.         protected AbstractHttpXmlDecoder(Class<?> clazz)
{
28.             this(clazz, false);
29.         }
30.
31.         protected AbstractHttpXmlDecoder(Class<?> clazz,
boolean isPrint) {
32.             this.clazz = clazz;
33.             this.isPrint = isPrint;
34.         }
35.
36.         protected Object decode0(ChannelHandlerContext
arg0, ByteBuf body)
37.             throws Exception {
38.             factory = BindingDirectory.getFactory(clazz);

39.             String content = body.toString(UTF_8);
```

```
40.         if (isPrint)
```

```
41.             System.out.println("The body is : " +  
content);
```

```
42.         reader = new StringReader(content);
```

```
43.             IUnmarshallingContext uctx =  
factory.createUnmarshallingContext();
```

44. *Object result = uctx.unmarshalDocument(reader);*

45. *reader.close();*

46. *reader = null;*

47. *return result;*

```

48.         }
49.         @Override
50.         public void exceptionCaught(ChannelHandlerContext
ctx, Throwable cause)
51.             throws Exception {
52.             // 清理
53.             if (reader != null) {
54.                 reader.close();
55.                 reader = null;
56.             }
57.         }
58.     }

```

38 47 HTTP 请求 解析 器 使用 JiBx 解析 XML 数据 并 使用 POJO 模型 表示 数据

53 55 使用 StringReader 解析 数据 并 使用 JVM 模型 表示 数据

3 HTTP+XML 请求 解析

POJO HTTP+XML
HTTP XML

10-13 HTTP+XML HTTP XML HttpXmlResponse

```
9.    public class HttpXmlResponse {
10.        private FullHttpResponse httpResponse;
11.        private Object result;
12.
13.        public HttpXmlResponse(FullHttpResponse
httpResponse,Object result){
14.            this.httpResponse = httpResponse;
15.            this.result = result;
16.        }
17.
18.        /**
19.         * @return the httpResponse
20.         */
21.        public final FullHttpResponse getHttpResponse() {
22.            return httpResponse;
23.        }
24.
25.        /**
26.         * @param httpResponse
27.         *         the httpResponse to set
```

```

28.         */
29.         public final void
setHttpResponse(FullHttpResponse httpResponse) {
30.         this.httpResponse = httpResponse;
31.     }
32.
33.     /**
34.      * @return the body
35.      */
36.     public final Object getResult() {
37.         return result;
38.     }
39.
40.     /**
41.      * @param body
42.      *         the body to set
43.      */
44.     public final void setResult(Object result) {
45.         this.result = result;
46.     }
47. }

```

FullHttpResponse
Object
Object
P0J0

□□□□□□□□□□XML□□□□□□

□□□□10-14 HTTP+XML □□□□□□□ HttpXmlResponseEncoder

```
17.    public class HttpXmlResponseEncoder extends
18.        AbstractHttpXmlEncoder<HttpXmlResponse> {
19.
20.        /*
21.            * (non-Javadoc)
22.            *
23.            * @see
24.                                     *
io.netty.handler.codec.MessageToMessageEncoder#encode(io.netty.
channel
25.            * .ChannelHandlerContext, java.lang.Object,
java.util.List)
26.        */
27.        protected void encode(ChannelHandlerContext
ctx,HttpXmlResponse msg,
28.            List<Object> out) throws Exception {
29.            ByteBuf body = encode0(ctx, msg.getResult());
30.            FullHttpResponse response =
msg.getHttpResponse();
31.            if (response == null) {
```

```

32.                                     response = new
DefaultFullHttpResponse(HTTP_1_1, OK, body);
33.     } else {
34.                                     response = new
DefaultFullHttpResponse(msg.getHttpResponse()
35.                                     .getProtocolVersion(),
msg.getHttpResponse().getStatus(),
36.                                     body);
37.     }
38.     response.headers().set(CONTENT_TYPE, "text/xml");
39.     setContentLength(response, body.readableBytes());
40.     out.add(response);
41. }
42. }

```

30 HTTP Netty DefaultFullHttpResponse
 content

API HTTP


```

out) throws Exception {
    25.         HttpXmlResponse resHttpXmlResponse = new
HttpXmlResponse(msg, decode0(
    26.             ctx, msg.content()));
    27.         out.add(resHttpXmlResponse);
    28.     }
    29. }

```

□ 25 □ □ □ DefaultFullHttpResponse □ HTTP □ □ □ □ □ □ □ □ □ P0J0 □ □ □ □
 HttpXmlResponse□□□□□□□□□□□□□□

5 HTTP+XML□□□□□

□□□□□□□□□

□1□□□HTTP□□□□□

□2□□□□□□□□□□□□□□XML□□□HTTP□□□□□□□□□

□3□□□HTTP□□□□□□□□□□XML□□□□□□□□□□□□□□P0J0□□□

4 HTTP

□ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □

10-16 HTTP+XML HttpXmlClient

[illegible]

37. `new HttpResponseDecoder());`

38. `ch.pipeline().addLast("http-
aggregator",`

39. `new HttpObjectAggregator(65536));`

40. `// XML`

41. `ch.pipeline().addLast(`

42. `"xml-decoder",`

43. `new`
`HttpXmlResponseDecoder(Order.class,`

44. `true));`

45. `ch.pipeline().addLast("http-encoder",`

46. `new HttpRequestEncoder());`

47. `ch.pipeline().addLast("xml-encoder",`

48. `new HttpXmlRequestEncoder());`

49. `ch.pipeline().addLast("xmlClientHandler",`

```

50.                                new HttpXmlClientHandle();

51.                                }
52.                                });
53.
54.                                // 连接成功
55.                                ChannelFuture f=b.connect(new
InetSocketAddress(port)).sync();
56.
57.                                // 关闭连接
58.                                f.channel().closeFuture().sync();
59.                                } finally {
60.                                // 关闭NIO线程组
61.                                group.shutdownGracefully();
62.                                }
63.                                }
64.
65.                                /**

```



```

22.
23.         @Override
                24.                             protected    void
messageReceived(ChannelHandlerContext ctx,
                25.                     HttpXmlResponse msg) throws Exception {
                26.                     System.out.println("The client receive response
of http header is : "
                27.                     + msg.getHttpResponse().headers().names());
                28.                     System.out.println("The client receive response
of http body is : "
                29.                     + msg.getResult());
                30.             }
                31.     }

```

12 HttpXmlRequest ChannelHandlerContext
 writeAndFlush HttpXmlRequest

24 30 HttpXmlResponse 28
 29 P0J0

```
2.    public class OrderFactory {
3.        public static Order create(long orderID) {
4.            Order order = new Order();
5.            order.setOrderNumber(orderID);
6.            order.setTotal(9999.999f);
7.            Address address = new Address();
8.            address.setCity(" ");
9.            address.setCountry(" ");
10.           address.setPostCode("123321");
11.           address.setState(" ");
12.           address.setStreet1(" ");
13.           order.setBillTo(address);
14.           Customer customer = new Customer();
15.           customer.setCustomerNumber(orderID);
16.           customer.setFirstName(" ");
17.           customer.setLastName(" ");
18.           order.setCustomer(customer);
19.           order.setShipping(Shipping.INTERNATIONAL_MAIL);
20.           order.setShipTo(address);
21.           return order;
22.       }
23.   }
```

6 HTTP+XML

HTTP

1 HTTP

2 HTTP XML POJO

3 POJO

4 HTTP+XML

5 HTTP

10-19 HTTP+XML HttpXmlServer

36.
aggregator",

ch.pipeline().addLast("http-

37.

new HttpObjectAggregator(65536));

38.

ch.pipeline()

39.

.addLast(

40. `"xml-decoder",`

41. `new HttpXmlRequestDecoder(`

42. `Order.class, true));`

43. `ch.pipeline().addLast("http-encoder",`

44. `new HttpResponseEncoder());`

45. `ch.pipeline().addLast("xml-encoder",`

46. `new HttpXmlResponseEncoder());`

47. `ch.pipeline().addLast("xmlServerHandler",`

48. `new HttpXmlServerHandler());`


```

49.         }
50.     });
51.         ChannelFuture future = b.bind(new
InetSocketAddress(port)). sync();
52.         System.out.println("HTTP服务器启动 : " +
"http://localhost:"
53.         + port);
54.         future.channel().closeFuture().sync();
55.     } finally {
56.         bossGroup.shutdownGracefully();
57.         workerGroup.shutdownGracefully();
58.     }
59. }
60.
61.     public static void main(String[] args) throws
Exception {
62.         int port = 8080;
63.         if (args.length > 0) {
64.             try {
65.                 port = Integer.parseInt(args[0]);
66.             } catch (NumberFormatException e) {
67.                 e.printStackTrace();
68.             }
69.         }
70.         new HttpXmlServer().run(port);

```

```

30.    public class XmlHttpRequestHandler extends
31.        SimpleChannelInboundHandler<HttpRequest> {
32.
33.        @Override
34.            public void messageReceived(final
ChannelHandlerContext ctx,
35.                HttpRequest request) throws Exception {
36.                HttpRequest req = request.getRequest();

```

37. *Order order = (Order) xmlRequest.getBody();*

38. *System.out.println("Http server receive request :
" + order);*

39. *dobusiness(order);*

40. *ChannelFuture future = ctx.writeAndFlush(new
HttpXmlResponse(null,*

```
41.         order));
```

```
42.    if (!isKeepAlive(request)) {
```

```
43.                                     future.addListener(new
GenericFutureListener<Future<? super Void>>() {
```

```
44.      public void operationComplete(Future future)
throws Exception {
```

```
45.         ctx.close();
```

```
46.     }
```

```
47.     });
```

```
48. }
```

```
49. }
```

```
50.
```

```
51.     private void dobusiness(Order order) {
```

```
52.         order.getCustomer().setFirstName("");
```

```
53.         order.getCustomer().setLastName("");
```

```
54.         List<String> midNames = new ArrayList<String>();
```

```

55.         midNames.add("   ");
56.         order.getCustomer().setMiddleNames(midNames);
57.         Address address = order.getBillTo();
58.         address.setCity("");
59.         address.setCountry("");
60.         address.setState("");
61.         address.setPostCode("123456");
62.         order.setBillTo(address);
63.         order.setShipTo(address);
64.     }
65.
66.     @Override
67.     public void exceptionCaught(ChannelHandlerContext
ctx, Throwable cause)
68.         throws Exception {
69.         cause.printStackTrace();
70.         if (ctx.channel().isActive()) {
71.             sendError(ctx, INTERNAL_SERVER_ERROR);
72.         }
73.     }
74.
75.         private static void
sendError(ChannelHandlerContext ctx,
76.             HttpResponseStatus status) {
77.             FullHttpResponse response = new
DefaultFullHttpResponse(HTTP_1_1,
78.                 status, Unpooled.copiedBuffer(" : " +

```

```

status.toString()
    79.                + "\r\n", CharsetUtil.UTF_8));
    80.        response.headers().set(CONTENT_TYPE,"text/plain;
charset=UTF-8");
                                                                    81.
ctx.writeAndFlush(response).addListener(ChannelFutureListener.
CLOSE);
    82.        }
    83.    }

```

消息接收方法 `messageReceived` 接收 `HttpXmlRequest` 消息，消息内容如下：

 37 消息内容如下：

 39 消息内容如下：

 40 47 消息内容如下：HTTP

70 71 消息内容如下：HTTP

HTTP+XML 消息内容如下：

10.3.5 HTTP+XML 消息

HTTP+XML 消息

用JiBxAntPOJOHTTP
用

1

The body is : <?xml version="1.0" encoding="UTF-8"?>
<order xmlns="http://phei.com/netty/protocol/http/xml/pojo"
orderNumber= "123" total="9999.999">
 <customer customerNumber="123">
 <firstName></firstName>
 <lastName></lastName>
 </customer>
 <billTo>
 <street1></street1>
 <city></city>
 <state></state>
 <postCode>123321</postCode>
 <country></country>
 </billTo>
 <shipping>INTERNATIONAL_MAIL</shipping>
 <shipTo>


```
<street1>□□□□</street1>
<city>□□□</city>
<state>□□□</state>
<postCode>123321</postCode>
<country>□□</country>
</shipTo>
</order>
```

□□□□□□□□□□□□□□

Http server receive request : Order [orderNumber=123, customer=Customer [customerNumber=123, firstName=□, lastName=□□, middleNames=null], billTo= Address [street1= □ □ □ □ , street2=null, city=□□□, state=□□□, postCode= 123321, country=□□], shipping=INTERNATIONAL_MAIL, shipTo=Address [street1=□□□□, street2=null, city=□□□, state=□□□, postCode=123321, country=□□], total=9999.999]

2□□□□

□□□□□□□□□□□□□□

The body is : <?xml version="1.0" encoding="UTF-8"?>

```
<order xmlns="http://phei.com/netty/protocol/http/xml/pojo"
orderNumber= "123" total="9999.999">
  <customer customerNumber="123">
    <firstName></firstName>
    <lastName></lastName>
    <middleName></middleName>
  </customer>
  <billTo>
    <street1></street1>
    <city></city>
    <state></state>
    <postCode>123456</postCode>
    <country></country>
  </billTo>
  <shipping>INTERNATIONAL_MAIL</shipping>
  <shipTo>
    <street1></street1>
    <city></city>
    <state></state>
    <postCode>123456</postCode>
    <country></country>
  </shipTo>
</order>
```

[illegible]

```
The client receive response of http body is : Order
[orderNumber=123,      customer=Customer      [customerNumber=123,
firstName= , lastName= , middleNames=[ ]], billTo=Address
[street1= , street2=null, city= , state= ,
postCode=123456, country= ], shipping=INTERNATIONAL_MAIL,
shipTo=Address [street1= , street2=null, city= , state= ,
postCode=123456, country= ], total=9999.999]
```

□□□□□□ HTTP+XML □□□□□□□□□□□□□□□□

10.3.6 ☐ ☐

HTTP+XML
API
HTTP+XML

10.4 □□

HTTPNettyHTTPHTTP+XML
NettyHTTP

HTTP+XML
API

第11章 WebSocket

本章主要介绍WebSocket协议/接口，以及相关的库。WebSocket是2005年提出的，它基于HTTP，但使用不同的端口，通过WebSocket API，可以实现全双工通信。

WebSocket是HTML5中实现服务器与客户端全双工通信的协议。它“模拟”Comet，但使用不同的端口，通过WebSocket API，可以实现全双工通信。WebSocket是基于HTTP的，但使用不同的端口，通过WebSocket API，可以实现全双工通信。WebSocket是基于Flash、XHR multipart、HTML Files、Gmail等技术的。

WebSocket是基于HTTP的，但使用不同的端口，通过WebSocket API，可以实现全双工通信。

WebSocket是基于Flash、XHR multipart、HTML Files、Gmail等技术的。WebSocket是基于Flash、XHR multipart、HTML Files、Gmail等技术的。

本章主要介绍WebSocket协议/接口，以及相关的库。

- HTTP
- WebSocket
- Netty WebSocket

11.1 HTTP

HTTP是超文本传输协议，它是万维网的数据通信的基础。

本章主要介绍WebSocket协议/接口，以及相关的库。WebSocket是2005年提出的，它基于HTTP，但使用不同的端口，通过WebSocket API，可以实现全双工通信。

2 HTTP HTTP

3

1 HTTP request HTTP request header

Comet Ajax Comet

HTTP HTML5 WebSocket WebSocket

11.2 WebSocket

WebSocket HTML5 WebSocket 2011 IETF RFC6455 WebSocket API W3C

WebSocket API WebSocket TCP HTTP

WebSocket

- TCP
-

- 使用Cookie
- 使用
- “ping/pong”
- 使用

11.2.1 WebSocket

WebSocket 是建立在Comet基础上的C/S架构，使用JavaScript的WebSocket接口，通过TCP和WebSocket协议，在TCP和Comet的基础上，使用Websocket.org的WebSocket接口，使用Web和WebSocket接口，使用11-1

11-1 WebSocket

WebSocket和Ajax，WebSocket和Web

11.2.2 WebSocket

11-2

11-2

WebSocket 11-3

11-3 WebSocket

浏览器通过WebSocket向服务器发起连接，服务器通过HTTP向浏览器返回响应，响应中包含“Upgrade: WebSocket”头，表示服务器支持WebSocket。浏览器收到响应后，通过HTTP向服务器发起WebSocket连接请求，服务器收到请求后，通过WebSocket向浏览器返回响应，响应中包含“Sec-WebSocket-Accept”头，表示服务器接受WebSocket连接。

图11-4 浏览器与服务器通过WebSocket连接

图11-4 WebSocket连接示意图

浏览器通过“Sec-WebSocket-Key”头向服务器发起连接，服务器收到请求后，通过SHA-1算法对“Sec-WebSocket-Key”头进行哈希运算，生成一个258EAF5E914-47DA-95CA-C5AB0DC85B11的哈希值，服务器将这个哈希值通过“Sec-WebSocket-Accept”头返回给浏览器。

11.2.3 WebSocket连接

浏览器通过“messages”头向服务器发起连接，服务器收到请求后，通过WebSocket向浏览器返回响应，响应中包含“messages”头，表示服务器支持WebSocket连接。

浏览器通过UTF-8[RFC3629]编码向服务器发起连接，服务器收到请求后，通过WebSocket向浏览器返回响应，响应中包含UTF-8[RFC3629]编码的响应数据。

WebSocket连接示意图11-5

图11-5 WebSocket连接示意图

11.2.4 WebSocket连接

WebSocket 和 TCP/TLS 的兼容性

TCP Close 和 TCP Close 的兼容性

WebSocket 和 Close 的兼容性

WebSocket 和 WebSocket 的兼容性

Netty 和 WebSocket 的兼容性

11.3 Netty WebSocket

Netty 和 HTTP 的兼容性

11.3.1 WebSocket

WebSocket 和 WebSocket 的兼容性

HTMLJSWebSocket“WebSocketWebSocket!”11-6

11-6 WebSocketHTML

WebSocket11-7WebSocket“WebSocket!”

11-7 WebSocket

11.3.2 WebSocket

WebSocketWebSocketHTTP 400 BAD REQUEST11-8

11-8 WebSocket

Socket11-9

11-9 WebSocket

HTTPHTTP

11-1 WebSocket WebSocketServer

```

1.    public class WebSocketServer {
2.        public void run(int port) throws Exception {
3.            EventLoopGroup bossGroup = new
NioEventLoopGroup();
4.            EventLoopGroup workerGroup = new
NioEventLoopGroup();
5.            try {
6.                ServerBootstrap b = new ServerBootstrap();
7.                b.group(bossGroup, workerGroup)
8.                    .channel(NioServerSocketChannel.class)
9.                    .childHandler(new
ChannelInitializer<SocketChannel>() {
10.
11.                @Override
12.                protected void initChannel(SocketChannel
ch)
13.                    throws Exception {
14.                    ChannelPipeline pipeline =
ch.pipeline();
15.                    pipeline.addLast("http-codec",

16.                    new HttpServerCodec());

```

17. `pipeline.addLast("aggregator",`

18. `new HttpObjectAggregator(65536));`

19. `ch.pipeline().addLast("http-chunked",`

20. `new ChunkedWriteHandler());`

```
21.                pipeline.addLast("handler",
```

```
22.                new WebSocketServerHandler());
```

```
23.                }
```

```
24.                });
```

```
25.
```

```
26.                Channel ch = b.bind(port).sync().channel();
```

```
27.                System.out.println("Web socket server started  
at port " + port
```

```
28.                + '.');
```

```
29.                System.out
```

```
30.                .println("Open your browser and navigate  
to http://localhost:"
```

```
31.                + port + '/');
```

```
32.
```

```
33.                ch.closeFuture().sync();
```

```
34.                } finally {
```

```

35.         bossGroup.shutdownGracefully();
36.         workerGroup.shutdownGracefully();
37.     }
38. }
39.
40.     public static void main(String[] args) throws
Exception {
41.         int port = 8080;
42.         if (args.length > 0) {
43.             try {
44.                 port = Integer.parseInt(args[0]);
45.             } catch (NumberFormatException e) {
46.                 e.printStackTrace();
47.             }
48.         }
49.         new WebSocketServer().run(port);
50.     }
51. }

```

15 16 `HttpServerCodec` HTTP
 17 18 `HttpObjectAggregator` HTTP
 19 20 `ChunkedWriteHandler`
 HTML5 `WebSocket` 21 22 `WebSocket`
`handler`

`WebSocket` `WebSocket`
 HTTP `ChannelPipeline` `WebSocket`

Handler 和 WebSocket 的类图如下所示
WebSocketServerHandler

图 11-2 WebSocket 的类图
WebSocketServerHandler

```
1.      public class WebSocketServerHandler extends
SimpleChannel InboundHandler<Object> {
2.          private static final Logger logger = Logger
3.
.getLogger(WebSocketServerHandler.class.getName());
4.
5.          private WebSocketServerHandshaker handshaker;
6.
7.          @Override
8.          public void messageReceived(ChannelHandlerContext
ctx, Object msg)
9.              throws Exception {
10.             // 处理HTTP
11.             if (msg instanceof FullHttpRequest) {
12.                 handleHttpRequest(ctx, (FullHttpRequest)
msg);
13.             }
14.             // WebSocket
15.             else if (msg instanceof WebSocketFrame) {
16.                 handleWebSocketFrame(ctx, (WebSocketFrame)
msg);
```



```

17.    }
18.    }
19.
20.    @Override
21.        public void
channelReadComplete(ChannelHandlerContext ctx) throws Exception
{
22.        ctx.flush();
23.    }
24.
25.        private void
handleRequest(ChannelHandlerContext ctx,
26.            FullHttpRequest req) throws Exception {
27.
28.        // 检查HTTP版本是否是HTTP1.1
29.        if (!req.getDecoderResult().isSuccess()
30.            ||
(!"websocket".equals(req.headers().get("Upgrade")))) {
31.            sendHttpResponse(ctx, req, new
DefaultFullHttpResponse(HTTP_1_1,
32.                BAD_REQUEST));
33.            return;
34.        }
35.
36.        // 创建WebSocketServerHandshakerFactory
37.        WebSocketServerHandshakerFactory wsFactory = new
WebSocketServerHandshakerFactory(

```

```

38.                "ws://localhost:8080/websocket", null,
false);
39.        handshaker = wsFactory.newHandshaker(req);
40.        if (handshaker == null) {
41.            WebSocketServerHandshakerFactory
42.
43.        } else {
44.            handshaker.handshake(ctx.channel(), req);
45.        }
46.        }
47.
48.                private void
handleWebSocketFrame(ChannelHandlerContext ctx,
49.            WebSocketFrame frame) {
50.
51.        // 检查是否是关闭帧
52.        if (frame instanceof CloseWebSocketFrame) {
53.            handshaker.close(ctx.channel(),
54.                (CloseWebSocketFrame) frame.retain());
55.            return;
56.        }
57.        // 检查是否是Ping帧
58.        if (frame instanceof PingWebSocketFrame) {
59.            ctx.channel().write(
60.                new
PongWebSocketFrame(frame.content().retain()));

```

```

61.         return;
62.     }
63.     // 检查是否是支持的帧类型
64.     if (!(frame instanceof TextWebSocketFrame)) {
65.         throw new
UnsupportedOperationException(String.format(
66.             "%s frame types not supported",
frame.getClass().getName()));
67.     }
68.
69.     // 获取请求
70.     String request = ((TextWebSocketFrame)
frame).text();
71.     if (logger.isLoggable(Level.FINE)) {
72.         logger.fine(String.format("%s received %s",
ctx.channel(), request));
73.     }
74.     ctx.channel().write(
75.         new TextWebSocketFrame(request
76.             + " , 基于Netty WebSocket实现"
77.             + new java.util.Date().toString()));
78.     }
79.
80.     private static void
sendHttpResponse(ChannelHandlerContext ctx,
81.         FullHttpRequest req, FullHttpResponse res) {
82.         // 检查是否是支持的帧类型

```

```

83.         if (res.getStatus().code() != 200) {
84.             ByteBuf buf =
Unpooled.copiedBuffer(res.getStatus().toString(),
85.                 CharsetUtil.UTF_8);
86.             res.content().writeBytes(buf);
87.             buf.release();
88.             setContentLength(res,
res.content().readableBytes());
89.         }
90.
91.         // 保持Keep-Alive
92.             ChannelFuture f =
ctx.channel().writeAndFlush(res);
93.         if (!isKeepAlive(req) || res.getStatus().code()
!= 200) {
94.             f.addListener(ChannelFutureListener.CLOSE);
95.         }
96.     }
97.
98.     @Override
99.     public void exceptionCaught(ChannelHandlerContext
ctx, Throwable cause)
100.         throws Exception {
101.         cause.printStackTrace();
102.         ctx.close();
103.     }
104. }

```

11 HTTP HTTP
handleHttpRequest WebSocket 29 34
Upgrade websocket HTTP
400

WebSocketServer
Handshaker WebSocket
ChannelPipeline WebSocket 11-10

11-10 WebSocket handler

WebSocket Encoder WebSocket Decoder
WebSocket handler WebSocket
WebSocket
ServerHandler WebSocketFrame 48 96
WebSocket
WebSocketServerHandshaker close WebSocket
Ping Pong WebSocket

TextWebSocketFrame
TextWebSocketFrame
TextWebSocketFrame TextWebSocketFrame

WebSocketServer.html

```
<html>
<head>
<meta charset="UTF-8">
Netty WebSocket 测试
</head>
<br>
<body>
<br>
<script type="text/javascript">
var
```

```
socket;
```

```
  if
```

```
(!window.WebSocket)
```

```
{
```

```
    window.WebSocket = window.MozWebSocket;
```

```
}
```

```
  if
```

```
(window.WebSocket) {  
    socket = new  
WebSocket("ws://localhost:8080/websocket");
```

```
    socket.onmessage = function(event) {
```

```
        var ta = document.getElementById('responseText');
```

```
        ta.value="";
```

```
ta.value = event.data
```

```
};
```

```
socket.onopen = function(event) {
```

```
var ta = document.getElementById('responseText');
```

```
ta.value = "WebSocket WebSocket!";
```



```
};
```

```
socket.onclose = function(event) {
```

```
var ta = document.getElementById('responseText');
```

```
ta.value = "";
```

```
ta.value = "WebSocket []!";
```

```
};
```

```
}
```

```
else
```

```
{
```

```
alert("[] [] [] [] [] [] [] [] WebSocket []!");
```

```
}
```

```
function
```

```
send(message) {  
    if (!window.WebSocket) { return; }  
  
    if (socket.readyState == WebSocket.OPEN) {  
  
        socket.send(message);  
  
    }  
    else
```

```
        {
            alert("WebSocket!!!!!!!!");
        }
    }
}
```

```
</script>
```

```
<form onsubmit="return false;">
```

```
<input type="text"
```

```
name="message"
```

```
value="Netty!!!!"
```

```
/>
```

```
<br><br>
```

```
<input type="button"
```

```
value="[]WebSocket[]"
```

```
onclick="send (this.form.message.value)"/>
```

```
<hr color="blue"
```

```
/>
```

```
<h3>[]</h3>
```

```
<textarea id="responseText"
```

```
style="width:500px;
```

```
height:300px;
```


11.4 实现

Netty 实现 HTTP 和 WebSocket 协议。Netty 实现 WebSocket 协议，需要实现 WebSocket 协议栈。Netty 实现 WebSocket 协议栈，需要实现 WebSocket 协议栈的每个层。

Netty 实现 WebSocket 协议栈，需要实现 Netty WebSocket 协议栈的每个层。Netty 实现 WebSocket 协议栈，需要实现 Netty WebSocket 协议栈的每个层。

Netty 实现 WebSocket 协议栈，需要实现 Netty WebSocket 协议栈的每个层。Netty 实现 WebSocket 协议栈，需要实现 Netty WebSocket 协议栈的每个层。Netty 实现 WebSocket 协议栈，需要实现 Netty WebSocket 协议栈的每个层。

Netty 实现 UDP 协议栈。

第12章 UDP 协议

UDP 即 User Datagram Protocol，即用户数据报协议。它是传输层协议，与 TCP 协议并列。UDP 协议是面向无连接的，而 TCP 协议是面向连接的。UDP 协议的数据报长度有限，而 TCP 协议的数据报长度没有限制。UDP 协议的数据报首部长度是固定的，而 TCP 协议的数据报首部长度是可变的。

UDP 协议的数据报首部长度是固定的，为 8 字节。UDP 协议的数据报首部包含源端口、目标端口、长度和校验和四个字段。

Netty 提供了对 UDP 协议的支持。

Netty 提供了对 UDP 协议的支持。

- UDP 协议
- UDP 协议
- UDP 协议
- UDP 协议

12.1 UDP 协议

UDP 协议是传输层协议，位于 OSI 模型的第 4 层。它使用 IP 协议作为网络层协议。UDP 协议的数据报首部长度是固定的，为 8 字节。UDP 协议的数据报首部包含源端口、目标端口、长度和校验和四个字段。

UDP 协议的数据报首部长度是固定的，为 8 字节。

1. 源端口：2 字节，范围 0-65535。

2 2 65535

3 2 UDP

4 2 UDP UDP “ ”
IP

Pseudo Header TCP UDP
IP IP IP TCP UDP
12
12-1

UDP 12-1

12-1 UDP

UDP

1 UDP UDP

2 UDP

3 UDP TCP UDP UDP
UDP TCP
TFTP TCP
FTP HTTP SMTP

Netty UDP

12.2 UDP服务器

UDP服务器和TCP服务器类似

图12-1 UDP服务器 ChineseProverbServer

```
1. public class ChineseProverbServer {
2.     public void run(int port) throws Exception {
3.         EventLoopGroup group = new NioEventLoopGroup();
4.         try {
5.             Bootstrap b = new Bootstrap();
6.
7.             b.group(group).channel(NioDatagramChannel.class)
8.
9.             .option(ChannelOption.SO_BROADCAST, true)
10.
11.             .handler(new
ChineseProverbServerHandler());
```

9.

```
b.bind(port).sync().channel().closeFuture().await();
10.      } finally {
11.          group.shutdownGracefully();
12.      }
13.  }
14.
15.      public static void main(String[] args) throws
Exception {
16.          int port = 8080;
17.          if (args.length > 0) {
18.              try {
19.                  port = Integer.parseInt(args[0]);
20.              } catch (NumberFormatException e) {
21.                  e.printStackTrace();
22.              }
23.          }
24.          new ChineseProverbServer().run(port);
25.      }
26.  }
```

□□□□ 6 □□□□□□ UDP □□□□□□ Channel □□□□□□□
NioDatagramChannel□□□□□□□Socket□□□□□□□□□□□□□□
handler□

□□□ TCP □□□ UDP □□□□□□□□□□□□□□□□□□□□□□□□□□
□ChannelPipeline□□□handler□□□□□□□□□□□□□□□□□□□□□□□□□□
□□

□□□ChineseProverbServerHandler□□□□

□□□□12-2 UDP□□□□□□ ChineseProverbServerHandler

```
1.    public class ChineseProverbServerHandler extends
2.        SimpleChannelInboundHandler<DatagramPacket> {
3.        // □□□□
4.        private static final String[] DICTIONARY={"□□□□□□
□□□□□□",
5.            "□□□□□□□□□□□□□□□□", "□□□□□□□□□□□□□□□□", "□□□□□
□□□□□□□□□□□□",
6.            "□□□□□□□□□□□□□□□□□!" };
7.
8.        private String nextQuote() {
9.                                int
quoteId=ThreadLocalRandom.current().nextInt(DICTIONARY.length);
10.        return DICTIONARY[quoteId];
11.    }
12.
```

```

13.         @Override
14.         public void messageReceived(ChannelHandlerContext
ctx, DatagramPacket packet)
15.             throws Exception {
16.                 String req =
packet.content().toString(CharsetUtil.UTF_8);
17.                 System.out.println(req);
18.                 if ("□□□□□?".equals(req)) {
19.                     ctx.writeAndFlush(new
DatagramPacket(Unpooled.copiedBuffer(
20.                         " □ □ □ □ □ □ : " + nextQuote(),
CharsetUtil.UTF_8), packet
21.                             .sender()));
22.                 }
23.             }
24.
25.         @Override
26.         public void exceptionCaught(ChannelHandlerContext
ctx, Throwable cause)
27.             throws Exception {
28.                 ctx.close();
29.                 cause.printStackTrace();
30.             }
31.     }

```

代码14 Netty UDP 接收数据并打印 Netty 的 `io.netty.channel.socket.DatagramPacket` 的 `16` 个 `packet` 属性。使用 `ByteBuf.toString(Charset)` 方法将字节数组转换为字符串。代码15 使用 `DatagramPacket` 的 `getAddress()` 方法获取 IP 地址。代码16 使用 `DatagramPacket` 的 `getLength()` 方法获取数据长度。

在 `ChineseProverbServerHandler` 类中，我们使用 `ThreadLocalRandom` 类来生成随机数。从 JDK7 开始，`ThreadLocalRandom` 类被引入到 `java.util.concurrent` 包中。

代码17 UDP 接收数据并打印 12-2

图12-2 UDP 接收数据并打印

12.3 UDP 客户端

UDP 客户端的实现非常简单。我们只需要创建一个 `UDPClient` 类，实现 `Runnable` 接口。在 `run()` 方法中，我们使用 `Socket` 类创建一个 `Socket` 对象，并指定要连接的 IP 地址和端口。

代码18 UDP 客户端实现

代码12-3 UDP 客户端实现 ChineseProverbClient

```
1. public class ChineseProverbClient {
2.
3.     public void run(int port) throws Exception {
4.         EventLoopGroup group = new NioEventLoopGroup();
5.         try {
```

```

6.          Bootstrap b = new Bootstrap();
                                                    7.
b.group(group).channel(NioDatagramChannel.class)
8.          .option(ChannelOption.SO_BROADCAST, true)
9.          .handler(new
ChineseProverbClientHandler());
10.         Channel ch = b.bind(0).sync().channel();
11.         // 向所有组播成员发送UDP包
12.         ch.writeAndFlush(
13.             new
DatagramPacket(Unpooled.copiedBuffer("组播包?"),
14.                 CharsetUtil.UTF_8), new
InetSocketAddress(
15.                     "255.255.255.255", port)).sync();
16.         if (!ch.closeFuture().await(15000)) {
17.             System.out.println("组播!");
18.         }
19.     } finally {
20.         group.shutdownGracefully();
21.     }
22. }
23.
24.     public static void main(String[] args) throws
Exception {
25.         int port = 8080;
26.         if (args.length > 0) {
27.             try {

```



```

28.         port = Integer.parseInt(args[0]);
29.     } catch (NumberFormatException e) {
30.         e.printStackTrace();
31.     }
32. }
33.     new ChineseProverbClient().run(port);
34. }
35. }

```

UDP Channel 是 Java 中用于接收 UDP 数据包的一个通道。UDP Channel 是 Java 中用于接收 UDP 数据包的一个通道。TCP 是用于接收 TCP 数据包的一个通道。handler 是用于处理数据包的一个类。

12-15 是 DatagramPacket 类的一个方法，用于设置 IP 地址。"255.255.255.255" 是一个特殊的 IP 地址，表示所有主机。15s 是一个时间间隔，表示 15 秒。handler 是一个用于处理数据包的方法。

handler 是一个用于处理数据包的方法。

12-4 UDP 通道 ChineseProverbClientHandler

```

1.     public class ChineseProverbClientHandler extends
2.         SimpleChannelInboundHandler<DatagramPacket> {
3.
4.         @Override
5.         public void messageReceived(ChannelHandlerContext
ctx, Datagram Packet msg)

```


图12-3 UDP数据包结构

图12-4 UDP数据包结构1

图12-4 UDP数据包结构1

图12-5 UDP数据包结构2

图12-5 UDP数据包结构2

UDP数据包结构图12-3展示了UDP数据包的结构。UDP数据包由16字节的头部和可变长度的数据部分组成。头部包含源端口、目标端口、长度和校验和。数据部分包含应用层数据。图12-4展示了UDP数据包的结构，其中头部包含源端口、目标端口、长度和校验和。数据部分包含应用层数据。图12-5展示了UDP数据包的结构，其中头部包含源端口、目标端口、长度和校验和。数据部分包含应用层数据。

12.5 Netty

Netty是一个高性能的异步事件驱动的网络应用程序框架，使用纯Java实现。它支持TCP、UDP和HTTP等协议。Netty的架构如图12-6所示。Netty的架构包括一个主线程（Boss线程）和多个工作线程（Worker线程）。Boss线程负责监听客户端连接，并将连接分配给Worker线程。Worker线程负责处理客户端请求，并将数据写入缓冲区。Netty的架构还支持多线程并发处理，提高了系统的吞吐量和性能。

Netty的架构如图12-6所示。Netty的架构包括一个主线程（Boss线程）和多个工作线程（Worker线程）。Boss线程负责监听客户端连接，并将连接分配给Worker线程。Worker线程负责处理客户端请求，并将数据写入缓冲区。Netty的架构还支持多线程并发处理，提高了系统的吞吐量和性能。

第13章 网络编程

本章主要介绍File、FileChannel、InputStream、OutputStream、Reader、Writer、NIO、FileChannel、Netty等网络编程相关的API和框架。

NIO是Java 1.4引入的，它提供了对非阻塞I/O的支持。

- 阻塞I/O：InputStream、OutputStream
- 非阻塞I/O：Reader、Writer

NIO提供了FileChannel接口，用于对文件进行非阻塞的I/O操作。Netty是一个高性能的NIO框架，它提供了对NIO的封装和扩展，使得网络编程更加简单和高效。

本章主要介绍以下内容：

- 阻塞I/O
- Netty框架
- Netty框架的扩展

13.1 网络编程

本章主要介绍Netty框架的使用，包括Netty框架的架构、Netty框架的扩展、Netty框架的集成等。

13.1.1 网络编程

[illegible]

```
Windows C:\windows Linux /home/lilinfeng
```

13.1.2 ☐ ☐ ☐ ☐

```

D:\java\nio\netty\FileServer.java
D:\java\nio\netty\FileServer.java

```

```

    Eclipse
D:\java\nio\nettyEclipse
classclassclass

```

13.1.3 ☐☐☐☐

[illegible]

doc windows office world Windows

Java

13.1.4 FileChannel

Java NIO FileChannel, JDK1.7 NIO1.0 FileChannel JDK1.7 NIO NIO2.0 AsynchronousFileChannel AIO

FileChannel FileChannel InputStream OutputStream RandomAccessFile FileChannel RandomAccessFile FileChannel

```
RandomAccessFile billFile = new
RandomAccessFile("home/lilinfeng/sms.bill", "rw");
FileChannel channel = billFile.getChannel();
```

FileChannel ByteBuffer FileChannel.read() int -1

FileChannel ByteBuffer FileChannel.write()

```
String content = "13888888888| | |VIP|";  
ByteBuffer writeBuffer = ByteBuffer.allocate(128);  
writeBuffer.put(content.getBytes());  
writeBuffer.flip();  
channel.write(buf);
```

FileChannel.close()

FileChannel.position(long pos)

13.2 Netty

FTP HTTP TCP
Socket+File
JVM

Netty NIO 10.3.3 JiBx
Ant POJO XML binding.xml

1 Netty 8080

2 CMD telnet TCP

3

4 4 CMD

5 CMD

13-1 FileServer

```
1 public class FileServer {
2     public void run(int port) throws Exception {
3         EventLoopGroup bossGroup = new NioEventLoopGroup();
4         EventLoopGroup workerGroup = new NioEventLoopGroup();
5         try {
6             ServerBootstrap b = new ServerBootstrap();
7             b.group(bossGroup, workerGroup)
8                 .channel(NioServerSocketChannel.class)
9                 .option(ChannelOption.SO_BACKLOG, 100)
10                .childHandler(new
ChannelInitializer<SocketChannel>() {
11                    /*
12                     * (non-Javadoc)
13                     *
14                     * @see
15                     *
16                     * io.netty.channel.ChannelInitializer#initChannel(io
17                     * .netty.channel.Channel)
18                     */
19                })
20            ;
21        } catch {
22            //
23        }
24    }
25 }
```

```
18    public void initChannel(SocketChannel ch)
19        throws Exception {
20        ch.pipeline().addLast(
21            new StringEncoder(CharsetUtil.UTF_8),

22            new LineBasedFrameDecoder(1024),

23            new StringDecoder(CharsetUtil.UTF_8),

24            new FileServerHandler());
```

```

25        }
26    });
27    ChannelFuture f = b.bind(port).sync();
28    System.out.println("Start file server at port : " +
port);
29    f.channel().closeFuture().sync();
30} finally {
31    // 清理
32    bossGroup.shutdownGracefully();
33    workerGroup.shutdownGracefully();
34}
35    }
36
37    public static void main(String[] args) throws
Exception {
38    int port = 8080;
39    if (args.length > 0) {
40        try {
41            port = Integer.parseInt(args[0]);
42        } catch (NumberFormatException e) {
43            e.printStackTrace();
44        }
45    }
46    new FileServer().run(port);
47    }
48}

```

22 ChannelPipeline LineBasedFrameDecoder4.3
23 StringDecoder
21 StringEncoder
binding.xml CMD

FileServerHandler

13-2 FileServerHandler

```
1  public class FileServerHandler extends
SimpleChannelInboundHandler<String> {
2
3      private static final String CR =
System.getProperty("line.separator");
4
5      /*
6       * (non-Javadoc)
7       *
8       * @see
9
io.netty.channel.SimpleChannelInboundHandler#messageReceived(io
.netty
10      * .channel.ChannelHandlerContext, java.lang.Object)
11      */
12      public void messageReceived(ChannelHandlerContext
ctx, String msg)
```

```

13    throws Exception {
14    File file = new File(msg);
15    if (file.exists()) {
16        if (!file.isFile()) {
17            ctx.writeAndFlush("Not a file : " + file + CR);
18            return;
19        }
20        ctx.write(file + " " + file.length() + CR);
21        RandomAccessFile randomAccessFile = new
RandomAccessFile(msg, "r");
22        FileRegion region = new DefaultFileRegion(
23            randomAccessFile.getChannel(), 0,
randomAccessFile.length());
24        ctx.write(region);
25        ctx.writeAndFlush(CR);
26        randomAccessFile.close();
27    } else {
28        ctx.writeAndFlush("File not found: " + file + CR);
29    }
30    }
31
32    /*
33     * (non-Javadoc)
34     *
35     * @see

```

36 *

io.netty.channel.ChannelHandlerAdapter#exceptionCaught(io.netty

```

. channel
    37    * .ChannelHandlerContext, java.lang.Throwable)
    38    */
    39    public void exceptionCaught(ChannelHandlerContext
ctx, Throwable cause)
    40        throws Exception {
    41        cause.printStackTrace();
    42        ctx.close();
    43    }
    44}

```

15~19行是异常处理，当遇到异常时，打印异常堆栈，并关闭通道。Netty 的 DefaultFileRegion 类提供了以下属性：

- FileChannel 文件通道
- Position 位置
- Count 计数

在 DefaultFileRegion 类中，ChannelHandlerContext 的 write 方法用于写入数据。Netty 的 CMD 类提供了以下属性：

13.3 Netty 应用

本节将介绍 Netty 的应用，包括 telnet 应用。图 13-1 展示了 telnet 应用的结构。

图 13-1 telnet 应用结构

□□□□□□□□□□□□□□□□“M:\software\eclipse-SDK-4.2.2-win32\ eclipse\workspace\book\binding.xml”□□□□□□□□□□□□
□□13-2□□□

□13-2 □□□□□□□□

13-3

□13-3 □□□□□□□□□□

Netty

13.4 ☐☐

```

NettyNetty
Netty

```

[illegible]

Netty ChunkedWriteHandler FileRegion API DOC

第14章 网络编程

本章主要介绍网络编程的相关知识，包括TCP/IP协议栈、Netty框架、NIO TCP编程等。

本章主要介绍网络编程的相关知识，包括TCP/IP协议栈、Netty框架、NIO TCP编程等。

本章主要介绍网络编程的相关知识。

- 网络编程
- Netty框架
- NIO TCP编程

14.1 网络编程

网络编程是指通过计算机网络进行数据通信的编程。本章主要介绍网络编程的相关知识，包括TCP/IP协议栈、Netty框架、NIO TCP编程等。

本章主要介绍网络编程的相关知识，包括TCP/IP协议栈、Netty框架、NIO TCP编程等。

本章主要介绍Java网络编程的相关知识。

1 RMI

2 Java Socket+Java

3 RPC Facebook Thrift Apache Avro

☐ HTTP+XML ☒ RESTful+JSON

☐ WebService

[illegible][illegible]

NettyTCPNetty
Netty

14.2 Netty

Netty TCP/IP HTTP

14.2.1 □□□□□

图14-1 Netty 的线程模型

Netty 的线程模型可以分为三个部分：Netty 的线程模型、Netty 的线程模型、Netty 的线程模型。Netty 的线程模型可以分为三个部分：Netty 的线程模型、Netty 的线程模型、Netty 的线程模型。Netty 的线程模型可以分为三个部分：Netty 的线程模型、Netty 的线程模型、Netty 的线程模型。

图14-1 Netty 的线程模型

14.2.2 线程模型

Netty 的线程模型可以分为三个部分：Netty 的线程模型、Netty 的线程模型、Netty 的线程模型。

1. Netty 的 NIO 线程模型

2. Netty 的 POJO 线程模型

3. Netty 的 IP 线程模型

4. Netty 的线程模型

5. Netty 的线程模型

14.2.3 线程模型

Netty 的线程模型可以分为三个部分：Netty 的线程模型、Netty 的线程模型、Netty 的线程模型。

图14-2 Netty 的线程模型

1. Netty 的 ID 线程模型

2. Netty 的 ID 线程模型

Netty 的 IP 线程模型

3

4

5

6

7

Netty TWO WAY ONE WAY Ping-Pong Ping Ping Ping Pong N Ping Ping Ping Ping T

14.2.4

Netty

-
-

14-1 Netty NettyMessage

14-2 Netty Header

14.2.5 Netty

14-3 Netty

14.2.6 Netty消息格式

1 Netty消息

Netty消息NettyMessage格式如下

1 crcCode java.nio.ByteBuffer.putInt(int value) 写入CRC码

2 length java.nio.ByteBuffer.putInt(int value) 写入长度

3 sessionID java.nio.ByteBuffer.putLong(long value) 写入会话ID

4 type: java.nio.ByteBuffer.put(byte b) 写入消息类型

5 priority java.nio.ByteBuffer.put(byte b) 写入消息优先级

6 attachment 消息附件——attachment 0 消息附件0
0 java.nio.ByteBuffer.putInt(0) 写入附件0
attachment 1——

- 写入附件大小
java.nio.ByteBuffer.putInt(attachment.size())
 - 写入Key 消息附件key byte
-

```
String key = null;
```

```
byte
```

```
[] value = null;
```

```
for
```

```
(Map.Entry<String, Object
```

```

> param : attachment.entrySet()) {
    key = param.getKey();
    buffer.writeString(key);
    value = marshaller.writeObject(param.getValue());
    buffer.writeBinary(value);
}
key = null;

```

```

value = null;

```

String → ByteBuffer → Jboss Marshalling →
 Object → byte → ...

7 → body → JBoss Marshalling → byte →
 java.nio.ByteBuffer.put(byte [] src) → ByteBuffer

...length →
 ByteBuffer

2 Netty 消息

NettyMessage 消息格式 java.nio.ByteBuffer Netty 消息格式

1 crcCode java.nio.ByteBuffer.getInt() 消息 CRC 码

2 length java.nio.ByteBuffer.getInt() Netty 消息长度

3 sessionID java.nio.ByteBuffer.getLong() 消息 ID

4 type java.nio.ByteBuffer.get() 消息类型

5 priority java.nio.ByteBuffer.get() 消息优先级

6 attachment 消息附件—— attachment java.nio.ByteBuffer.getInt() 消息附件数量 0 消息附件数量 for

String key = null;

Object

`value = null;`

for

(int

`i = 0; i < size; i++) {`

`key = buffer.readString();`

`value =`

`unmarshaller.readObject(buffer.readBinary());`

this.

Netty 通过 `ChannelHandler` 接口实现 `A` 和 `B` 的 `ChannelHandler` 接口，`A` 和 `B` 的 `ChannelHandler` 接口实现 `ChannelHandler` 接口。

通过 `ChannelHandler` 接口实现 `IP` 的 `ChannelHandler` 接口，`IP` 的 `ChannelHandler` 接口实现 `ChannelHandler` 接口。

通过 `ChannelHandler` 接口实现 `ChannelHandler` 接口。

1. `type` 为 3

2. `type` 为 0

3. `type` 为 0

4. `type` 为 22

通过 `ChannelHandler` 接口实现 `IP` 的 `ChannelHandler` 接口。

1. `type` 为 4

2. `type` 为 0

3. `byte` 为 0 到 1

通过 `ChannelHandler` 接口实现 `ChannelHandler` 接口。

14.2.8 总结

14.2.10 网络层

Netty 支持 IP 地址的绑定和监听。通过 `ChannelInitializer` 的 `initChannel` 方法，可以配置 `Channel` 的 `socketOptions`，如 `setSoLinger`、`setSoKeepAlive` 等。

Netty 支持 SSL/TLS 加密。通过 `ChannelInitializer` 的 `initChannel` 方法，可以配置 `Channel` 的 `sslContext`。

Netty 支持 IP 地址的绑定和监听。

14.2.11 应用层

Netty 支持应用层的编解码。通过 `ChannelInitializer` 的 `initChannel` 方法，可以配置 `Channel` 的 `encoder` 和 `decoder`。Netty 还支持 `attachment` 功能，可以在 `ChannelHandlerContext` 中存储任意对象。

Netty 支持应用层的拦截器。通过 `ChannelInitializer` 的 `initChannel` 方法，可以配置 `Channel` 的 `handler`。Netty 还支持 `Servlet`、`Filter Chain`、`AOP` 等功能。

14.3 Netty 应用

14.3.1 网络层

Netty 支持网络层的编解码。通过 `ChannelInitializer` 的 `initChannel` 方法，可以配置 `Channel` 的 `encoder` 和 `decoder`。

图 14-1 NettyMessage 类

```
1 public final class NettyMessage {
2     private Header header; //[]
3     private Object body; //[]
4
5     /**
6      * @return the header
7      */
8     public final Header getHeader() {
9         return header;
10    }
11
12    /**
13     * @param header
14     *         the header to set
15     */
16    public final void setHeader(Header header) {
17        this.header = header;
18    }
19
20    /**
21     * @return the body
22     */
23    public final Object getBody() {
24        return body;
25    }
26
27    /**
```



```

28    * @param body
29    *           the body to set
30    */
31    public final void setBody(Object body) {
32        this.body = body;
33    }
34
35    /*
36    * (non-Javadoc)
37    *
38    * @see java.lang.Object#toString()
39    */
40    @Override
41    public String toString() {
42        return "NettyMessage [header=" + header + "];
43    }
44}

```

14-2 Header

```

1    public final class Header {
2        private int crcCode = 0xabef0101;
3        private int length; //
4        private long sessionId; // ID
5        private byte type; //
6        private byte priority; //

```

```

7      private Map<String, Object> attachment = new
HashMap<String, Object>(); //
8
9      /**
10     * @return the crcCode
11     */
12     public final int getCrcCode() {
13     return crcCode;
14     }
15
16     /**
17     * @param crcCode
18     *         the crcCode to set
19     */
20     public final void setCrcCode(int crcCode) {
21     this.crcCode = crcCode;
22     }
23
24     /**
25     * @return the length
26     */
27     public final int getLength() {
28     return length;
29     }
30
31     /**
32     * @param length

```

```
33□    *           the length to set
34□    */
35□    public final void setLength(int length) {
36□this.length = length;
37□    }
38□
39□    /**
40□    * @return the sessionID
41□    */
42□    public final long getSessionID() {
43□return sessionID;
44□    }
45□
46□    /**
47□    * @param sessionID
48□    *           the sessionID to set
49□    */
50□    public final void setSessionID(long sessionID) {
51□this.sessionID = sessionID;
52□    }
53□
54□    /**
55□    * @return the type
56□    */
57□    public final byte getType() {
58□return type;
59□    }
```

```
60
61  /**
62   * @param type
63   *           the type to set
64   */
65   public final void setType(byte type) {
66   this.type = type;
67   }
68
69  /**
70   * @return the priority
71   */
72   public final byte getPriority() {
73   return priority;
74   }
75
76  /**
77   * @param priority
78   *           the priority to set
79   */
80   public final void setPriority(byte priority) {
81   this.priority = priority;
82   }
83
84  /**
85   * @return the attachment
86   */
```

```

87    public final Map<String, Object> getAttachment() {
88    return attachment;
89    }
90
91    /**
92     * @param attachment
93     *         the attachment to set
94     */
95    public final void setAttachment(Map<String, Object>
attachment) {
96    this.attachment = attachment;
97    }
98
99    /*
100     * (non-Javadoc)
101     *
102     * @see java.lang.Object#toString()
103     */
104    @Override
105    public String toString() {
106    return "Header [crcCode=" + crcCode + ", length=" +
length
107        + ", sessionId=" + sessionId + ", type=" + type
+ ", priority="
108        + priority + ", attachment=" + attachment + "];"
109    }
110}

```

NettyMessage 的 encode 方法，
返回一个 List<Object> 类型的集合。

14.3.2 实现

实现 NettyMessageDecoder 和 NettyMessageEncoder 两个
NettyMessage 的子接口。

图 14-3 Netty 的 NettyMessageEncoder

```
1 public final class NettyMessageEncoder extends
2     MessageToMessageEncoder<NettyMessage> {
3
4     MarshallingEncoder marshallEncoder;
5
6     public NettyMessageEncoder() throws IOException {
7         this.marshallEncoder = new MarshallingEncoder();
8     }
9
10    @Override
11        protected void encode(ChannelHandlerContext ctx,
NettyMessage msg,
12        List<Object> out) throws Exception {
13
14        if (msg == null || msg.getHeader() == null)
15            throw new Exception("The encode message is null");
```

```

16[]ByteBuf sendBuf = Unpooled.buffer();
17[]sendBuf.writeInt((msg.getHeader().getCrcCode()));
18[]sendBuf.writeInt((msg.getHeader().getLength()));
19[]sendBuf.writeLong((msg.getHeader().getSessionID()));
20[]sendBuf.writeByte((msg.getHeader().getType()));
21[]sendBuf.writeByte((msg.getHeader().getPriority()));
                                     22           []
sendBuf.writeInt((msg.getHeader().getAttachment().size()));
23[]String key = null;
24[]byte[] keyArray = null;
25[]Object value = null;
        26 [] for (Map.Entry<String, Object> param :
msg.getHeader().getAttachment()
27[]     .entrySet()) {
28[]     key = param.getKey();
29[]     keyArray = key.getBytes("UTF-8");
30[]     sendBuf.writeInt(keyArray.length);
31[]     sendBuf.writeBytes(keyArray);
32[]     value = param.getValue();
33[]     marshallngEncoder.encode(value, sendBuf);
34[]}
35[]key = null;
36[]keyArray = null;
37[]value = null;
38[]if (msg.getBody() != null) {
39[]     marshallngEncoder.encode(msg.getBody(), sendBuf);
40[]} else

```

```

41    sendBuf.writeInt(0);
42    sendBuf.setInt(4, sendBuf.readableBytes());
43    }
44}

```

14-4 Netty MarshallingEncoder

```

1 public class MarshallingEncoder {
2     private static final byte[] LENGTH_PLACEHOLDER = new
byte[4];
3     Marshaller marshaller;
4
5     public MarshallingEncoder() throws IOException {
6         marshaller =
MarshallingCodecFactory.buildMarshalling();
7     }
8
9     protected void encode(Object msg, ByteBuf out) throws
Exception {
10    try {
11        int lengthPos = out.writerIndex();
12        out.writeBytes(LENGTH_PLACEHOLDER);
13        ChannelBufferByteOutput output = new
ChannelBufferByteOutput(out);
14        marshaller.start(output);
15        marshaller.writeObject(msg);

```



```

16    marshaller.finish();
17    out.setInt(lengthPos, out.writerIndex() - lengthPos
- 4);
18} finally {
19    marshaller.close();
20}
21    }
22}

```

14-5 NettyNettyMessageDecoder

```

1    public class NettyMessageDecoder extends
LengthFieldBasedFrameDecoder {
2
3    MarshallingDecoder marshallDecoder;
4
5    public NettyMessageDecoder(int maxFrameLength, int
lengthFieldOffset,
6    int lengthFieldLength) throws IOException {
7        super(maxFrameLength, lengthFieldOffset,
lengthFieldLength);
8    marshallDecoder = new MarshallingDecoder();
9    }
10
11    @Override
12    protected Object decode(ChannelHandlerContext ctx,

```

ByteBuf in)

```
13    throws Exception {
14    ByteBuf frame = (ByteBuf) super.decode(ctx, in);
15    if (frame == null) {
16        return null;
17    }
18
19    NettyMessage message = new NettyMessage();
20    Header header = new Header();
21    header.setCrcCode(in.readInt());
22    header.setLength(in.readInt());
23    header.setSessionID(in.readLong());
24    header.setType(in.readByte());
25    header.setPriority(in.readByte());
26
27    int size = in.readInt();
28    if (size > 0) {
29        Map<String, Object> attch = new HashMap<String,
Object>(size);
30        int keySize = 0;
31        byte[] keyArray = null;
32        String key = null;
33        for (int i = 0; i < size; i++) {
34            keySize = in.readInt();
35            keyArray = new byte[keySize];
36            in.readBytes(keyArray);
37            key = new String(keyArray, "UTF-8");
```

```

38    attch.put(key, marshallDecoder.decode(in));
39    }
40    keyArray = null;
41    key = null;
42    header.setAttachment(attch);
43    }
44    if (in.readableBytes() > 4) {
45        message.setBody(marshallDecoder.decode(in));
46    }
47    message.setHeader(header);
48    return message;
49    }
50    }

```

Netty的LengthFieldBasedFrameDecoder类是Netty
 TCP连接中用于解码帧的类。Netty
 的LengthFieldBasedFrameDecoder类
 是I/O模型中的14-3

14-3 类图

14-6 Netty的MarshallDecoder

```

1    public class MarshallDecoder {
2        private final Unmarshaller unmarshaller;
3

```

```

4    /**
5    * Creates a new decoder whose maximum object size is
{@code 1048576} bytes.
6    * If the size of the received object is greater than
{@code 1048576} bytes,
7    * a {@link StreamCorruptedException} will be raised.
8    *
9    * @throws IOException
10   *
11   */
12   public MarshallingDecoder() throws IOException {
13       unmarshaller =
MarshallingCodecFactory.buildUnMarshalling();
14   }
15
16   protected Object decode(ByteBuf in) throws Exception
{
17       int objectSize = in.readInt();
18       ByteBuf buf = in.slice(in.readerIndex(), objectSize);
19       ByteInput input = new ChannelBufferByteInput(buf);
20       try {
21           unmarshaller.start(input);
22           Object obj = unmarshaller.readObject();
23           unmarshaller.finish();
24           in.readerIndex(in.readerIndex() + objectSize);
25           return obj;
26       } finally {

```

```

27    unmarshaller.close();
28}
29    }
30}

```

14.2.6 在 `JBoss Marshall` 中，`JDK` 的 `ByteBuffer` 和 `JBoss Marshall` 的 `ByteBuffer` 是不同的。

14.3.3 通道处理器

在 `TCP` 中，通道处理器（`ChannelHandler`）是一个接口，它定义了通道处理器的行为。

在 `ChannelHandler` 接口中，`fireChannelActive()` 方法用于通知通道已经激活。

14-7 LoginAuthReqHandler

```

1  public class LoginAuthReqHandler extends
ChannelHandlerAdapter {
2
3      /**
4          * Calls {@link
ChannelHandlerContext#fireChannelActive()} to forward to the
5          * next {@link ChannelHandler} in the {@link

```

ChannelPipeline}.

```
6   *  
7   * Sub-classes may override this method to change  
behavior.
```

```
8   */  
9   @Override  
10  public void channelActive(ChannelHandlerContext  
ctx) throws Exception {
```

```
11  ctx.writeAndFlush(buildLoginReq());  
12  }  
13  
14  /**  
15      * Calls {@link  
ChannelHandlerContext#fireChannelRead(Object)} to forward to  
16      * the next {@link ChannelHandler} in the {@link  
ChannelPipeline}.
```

```
17  *  
18  * Sub-classes may override this method to change  
behavior.
```

```
19  */  
20  @Override  
21  public void channelRead(ChannelHandlerContext ctx,  
Object msg)
```

```
22  throws Exception {  
23  NettyMessage message = (NettyMessage) msg;  
24  
25  //   

```

```

26 if (message.getHeader() != null
    27     && message.getHeader().getType() ==
MessageType.LOGIN_RESP
28     .value()) {
29     byte loginResult = (byte) message.getBody();
30     if (loginResult != (byte) 0) {
31         // 
32         ctx.close();
33     } else {
34         System.out.println("Login is ok : " + message);
35         ctx.fireChannelRead(msg);
36     }
37 } else
38     ctx.fireChannelRead(msg);
39 }
40
41 private NettyMessage buildLoginReq() {
42     NettyMessage message = new NettyMessage();
43     Header header = new Header();
44     header.setType(MessageType.LOGIN_REQ.value());
45     message.setHeader(header);
46     return message;
47 }
48
49     public void exceptionCaught(ChannelHandlerContext
ctx, Throwable cause)
50     throws Exception {

```


ChannelPipeline}.

```
8   *
9   * Sub-classes may override this method to change
behavior.
10  */
11  @Override
12  public void channelRead(ChannelHandlerContext ctx,
Object msg)
13  throws Exception {
14  NettyMessage message = (NettyMessage) msg;
15
16  // 验证消息头
17  if (message.getHeader() != null
18      && message.getHeader().getType() ==
MessageType.LOGIN_REQ
19      .value()) {
20      String nodeIndex =
ctx.channel().remoteAddress().toString();
21      NettyMessage loginResp = null;
22      // 验证节点
23      if (nodeCheck.containsKey(nodeIndex)) {
24          loginResp = buildResponse((byte) -1);
25      } else {
26          InetSocketAddress address = (InetSocketAddress)
ctx.channel()
27              .remoteAddress();
28          String ip = address.getAddress().getHostAddress();
```

```

29[]    boolean isOK = false;
30[]    for (String WIP : whitekList) {
31[]        if (WIP.equals(ip)) {
32[]            isOK = true;
33[]            break;
34[]        }
35[]    }
36[]    loginResp = isOK ? buildResponse((byte) 0)
37[]        : buildResponse((byte) -1);
38[]    if (isOK)
39[]        nodeCheck.put(nodeIndex, true);
40[]    }
41[]    System.out.println("The login response is : " +
loginResp
42[]        + " body [" + loginResp.getBody() + "]");
43[]    ctx.writeAndFlush(loginResp);
44[]} else {
45[]    ctx.fireChannelRead(msg);
46[]}
47[]    }
48[]
49[]    private NettyMessage buildResponse(byte result) {
50[]NettyMessage message = new NettyMessage();
51[]Header header = new Header();
52[]header.setType(MessageType.LOGIN_RESP.value());
53[]message.setHeader(header);
54[]message.setBody(result);

```

```

55    return message;
56    }
57
58    public void exceptionCaught(ChannelHandlerContext
ctx, Throwable cause)
59    throws Exception {
60
61    ctx.close();
62    ctx.fireExceptionCaught(cause);
63    }
64}

```

```

    23  IP 1747
    /127.0.0.1:12088
    ChannelHandlerContext  Channel
    InetSocketAddress
    buildResponse

```

```

    
```

14.3.4

1. 心跳检测

2. 心跳检测

14-9 HeartBeatReqHandler

```

1  public class HeartBeatReqHandler extends
ChannelHandlerAdapter {
2      private volatile ScheduledFuture<?> heartBeat;
3
4      @Override
5      public void channelRead(ChannelHandlerContext ctx,
Object msg)
6          throws Exception {
7      NettyMessage message = (NettyMessage) msg;
8      // 心跳检测
9      if (message.getHeader() != null
10         && message.getHeader().getType() ==
MessageType.LOGIN_RESP
11         .value()) {
12      heartBeat = ctx.executor().scheduleAtFixedRate(
13         new HeartBeatReqHandler.HeartBeatTask(ctx), 0,
14         5000,
15         TimeUnit.MILLISECONDS);
16      } else if (message.getHeader() != null
17         && message.getHeader().getType() ==
```

```

MessageType.HEARTBEAT_RESP
17[]         .value()) {
18[]     System.out
19[]         .println("Client receive server heart beat
message : ---> "
20[]         + message);
21[] } else
22[]     ctx.fireChannelRead(msg);
23[]     }
24[]
25[]     private class HeartBeatTask implements Runnable {
26[] private final ChannelHandlerContext ctx;
27[]
28[] public HeartBeatTask(final ChannelHandlerContext ctx) {
29[]     this.ctx = ctx;
30[] }
31[]
32[] @Override
33[] public void run() {
34[]     NettyMessage heatBeat = buildHeatBeat();
35[]     System.out
36[]         .println("Client send heart beat message to
server : ---> "
37[]         + heatBeat);
38[]     ctx.writeAndFlush(heatBeat);
39[] }
40[]

```

```

41 private NettyMessage buildHeartBeat() {
42     NettyMessage message = new NettyMessage();
43     Header header = new Header();
44     header.setType(MessageType.HEARTBEAT_REQ.value());
45     message.setHeader(header);
46     return message;
47 }
48 }
49
50 @Override
51     public void exceptionCaught(ChannelHandlerContext
ctx, Throwable cause)
52     throws Exception {
53     if (heartBeat != null) {
54         heartBeat.cancel(true);
55         heartBeat = null;
56     }
57     ctx.fireExceptionCaught(cause);
58 }
59 }

```

9. `HeartBeatReqHandler` 类中，`NioEventLoop` 的 `schedule` 方法，每隔 5000 毫秒，每隔 5 秒，向客户端发送心跳包。

handler 15 20

HeartBeatTask ChannelHandlerContext

handler

14-10 HeartBeatRespHandler

```
1 public class HeartBeatRespHandler extends
ChannelHandlerAdapter {
2     @Override
3     public void channelRead(ChannelHandlerContext ctx,
Object msg)
4         throws Exception {
5         NettyMessage message = (NettyMessage) msg;
6         //
7         if (message.getHeader() != null
8             && message.getHeader().getType() ==
MessageType.HEARTBEAT_REQ
9             .value()) {
10            System.out.println("Receive client heart beat
message : ---> "
11            + message);
12            NettyMessage heartBeat = buildHeartBeat();
13            System.out
```


14-4

14-4

Channel Bootstrap
closeFuture 5s

14.3.6

14-11 NettyClient

```
1 public class NettyClient {
2     private ScheduledExecutorService executor = Executors
3         .newScheduledThreadPool(1);
4     EventLoopGroup group = new NioEventLoopGroup();
5     public void connect(int port, String host) throws
Exception {
6         // NIO
7         try {
8             Bootstrap b = new Bootstrap();
9             b.group(group).channel(NioSocketChannel.class)
10                .option(ChannelOption.TCP_NODELAY, true)
11                .handler(new ChannelInitializer<SocketChannel>())
```

```

{
    12    @Override
    13    public void initChannel(SocketChannel ch)
    14        throws Exception {
    15        ch.pipeline().addLast(
    16            new NettyMessageDecoder(1024 * 1024, 4,
4));
    17        ch.pipeline().addLast("MessageEncoder",
    18            new NettyMessageEncoder());
    19    ch.pipeline().addLast("readTimeoutHandler",
    20        new ReadTimeoutHandler(50));
    21        ch.pipeline().addLast("LoginAuthHandler",
    22            new LoginAuthReqHandler());
    23        ch.pipeline().addLast("HeartBeatHandler",
    24            new HeartBeatReqHandler());
    25    }
    26    });
    27    // 连接成功
    28    ChannelFuture future = b.connect(
    29        new InetSocketAddress(host, port),
    30        new InetSocketAddress(NettyConstant.LOCALIP,
    31            NettyConstant.LOCAL_PORT)).sync();
    32    future.channel().closeFuture().sync();
    33    } finally {
    34        // 关闭连接
    35        executor.execute(new Runnable() {
    36            @Override

```

```

37    public void run() {
38        try {
39            TimeUnit.SECONDS.sleep(5);
40            try {
41                connect(NettyConstant.PORT,
NettyConstant.REMOTEIP);// 连接成功
42            } catch (Exception e) {
43                e.printStackTrace();
44            }
45            } catch (InterruptedException e) {
46                e.printStackTrace();
47            }
48        }
49    });
50}
51    }
52
53    /**
54     * @param args
55     * @throws Exception
56     */
57    public static void main(String[] args) throws
Exception {
58        new NettyClient().connect(NettyConstant.PORT,
NettyConstant.REMOTEIP);
59    }
60}

```

15 16 NettyMessageDecoder Netty
17 18 Netty
Handler Handler Handler

28 TCP
TCP

Netty ChannelPipeline ChannelHandler
Handler AOP Handler Chain
AOP

14.3.7

14-12 NettyServer

```
1 public class NettyServer {
2     public void bind() throws Exception {
3         // NIO
4         EventLoopGroup bossGroup = new NioEventLoopGroup();
5         EventLoopGroup workerGroup = new NioEventLoopGroup();
6         ServerBootstrap b = new ServerBootstrap();
7         b.group(bossGroup,
```

```

workerGroup).channel(NioServerSocketChannel.class)
    8    .option(ChannelOption.SO_BACKLOG, 100)
    9    .handler(new LoggingHandler(LogLevel.INFO))
        10    .childHandler(new
ChannelInitializer<SocketChannel>() {
    11        @Override
    12        public void initChannel(SocketChannel ch)
    13            throws IOException {
    14            ch.pipeline().addLast(
    15                new NettyMessageDecoder(1024 * 1024, 4,
4));
        16            ch.pipeline().addLast(new
NettyMessageEncoder());
    17    ch.pipeline().addLast("readTimeoutHandler",
    18        new ReadTimeoutHandler(50));
        19            ch.pipeline().addLast(new
LoginAuthRespHandler());
    20            ch.pipeline().addLast("HeartBeatHandler",
    21                new HeartBeatRespHandler());
    22        }
    23    });
    24
    25    // 启动服务器
        26    b.bind(NettyConstant.REMOTEIP,
NettyConstant.PORT).sync();
    27    System.out.println("Netty server start ok : "
    28        + (NettyConstant.REMOTEIP + " : " +

```

```

NettyConstant.PORT));
    29    }
    30
    31    public static void main(String[] args) throws
Exception {
    32    new NettyServer().bind();
    33    }
    34}

```

ChannelPipelineNettyLoginAuthRespHandlerHeartBeatRespHandler

14.4

14.4.1

5s

14-5

14-5

14-6

14-6

14-7

14-7 TCP

14.4.2 □□□□□□□□□□

□ □

0100000000000000

□2□□□□□□□□□□

3

4

[illegible]

□6□□□□□□□□□□□□□□□□

7

14-8

□14-8 □□□□□□□□

□□□□□□□14-9□□□

□14-9 □□□□□□□□□□

14-10

□14-10 □□□□□□□□

14-11

图14-11 阻塞IO模型

阻塞IO模型示意图14-12

图14-12 阻塞IO模型

阻塞IO模型示意图14-13

图14-13 阻塞IO模型

使用netstat查看TCP连接状态14-14

图14-14 TCP连接状态

14.4.3 非阻塞IO模型

非阻塞IO模型示意图14-15

非阻塞IO模型示意图14-15

图14-15 非阻塞IO模型

非阻塞IO模型示意图14-15

14.5 总结

本文介绍了四种IO模型：阻塞IO、非阻塞IO、多路复用IO和异步IO。其中，阻塞IO模型是最简单的，但效率最低；非阻塞IO模型效率较高，但编程复杂；多路复用IO模型效率最高，但编程最复杂；异步IO模型效率最高，但编程最复杂。

Netty是一个高性能的异步IO框架，它支持多种IO模型，包括阻塞IO、非阻塞IO、多路复用IO和异步IO。Netty的编程模型非常简单，易于学习和使用。

Netty 4.1.0.Final

Netty 4.1.0.Final
Netty 4.1.0.Final

Demo
Netty 4.1.0.Final

📖📖📖 **Netty**📖📖📖📖📖📖

📖15📖 ByteBuf📖📖📖📖

📖16📖 Channel📖Unsafe

📖17📖 ChannelPipeline📖ChannelHandler

📖18📖 EventLoop📖EventLoopGroup

📖19📖 Future📖Promise

第15章 ByteBuf 的应用

本章主要介绍 Netty NIO 的应用 API。本章主要介绍 Netty 的应用 API，包括 Netty 的应用 API 和 Netty 的应用 API。本章主要介绍 Netty 的应用 API，包括 Netty 的应用 API 和 Netty 的应用 API。

本章主要介绍

- ByteBuf 的应用
- ByteBuf 的应用
- ByteBuf 的应用

15.1 ByteBuf 的应用

本章主要介绍 JDK NIO 的应用 API。本章主要介绍 JDK NIO 的应用 API，包括 java.nio.Buffer 的应用 API。

图 15-1 java.nio.Buffer 的应用

本章主要介绍 Boolean 的应用 API。本章主要介绍 Boolean 的应用 API，包括 Boolean 的应用 API 和 Boolean 的应用 API。

本章主要介绍 ByteBuffer 的应用 API。本章主要介绍 ByteBuffer 的应用 API，包括 ByteBuffer 的应用 API 和 ByteBuffer 的应用 API。

本章主要介绍 position 的应用 API。本章主要介绍 position 的应用 API，包括 position 的应用 API 和 position 的应用 API。

第3章 ByteBuffer API 提供了非常丰富的 API 来操作 ByteBuffer，本章将介绍 ByteBuffer 的 API。

本章将介绍 Netty 中的 ByteBuffer 和 ByteBuf 的区别，以及 ByteBuf 和 ByteBuffer 的转换。

15.1.1 ByteBuf 简介

ByteBuf 是 Netty 中用于操作字节数据的接口，它提供了比 ByteBuffer 更丰富的 API。Netty ByteBuf 的 API 如下：

ByteBuf 是 Netty 中用于操作字节数据的接口，它提供了比 ByteBuffer 更丰富的 API。ByteBuf 的 API 如下：

- 7 个 Java 包中的 byte 类型 ByteBuffer 和 ByteBuf 的 API。
- 提供了 copy 和 slice 方法。
- 提供了 read 方法。
- 提供了 write 方法。
- 提供了 clear 方法。

Netty ByteBuf 的 API 如下：

- Netty ByteBuf 的 API 如下：
- Netty ByteBuf 的 API 如下：

Netty ByteBuf 的 API 如下：

```
ByteBuffer buffer = ByteBuffer.allocate(88);  
String value = "Netty";  
buffer.put(value.getBytes());  
buffer.flip();  
byte
```

```
[] vArray = new byte
```

```
[buffer.remaining()];  
    buffer.get(vArray);  
    String decodeValue = new
```

```
String(vArray);
```

```
    flip()
```

图15-2 ByteBuffer flip()

图15-2 调用 `flip()` 方法后 `position` 和 `capacity` 的变化

调用 `flip()` 方法后，`limit` 被设置为 `position`，`position` 被设置为 0，`capacity` 保持不变。调用 `flip()` 方法后，`position` 和 `limit` 的值如图15-3所示。

图15-3 `ByteBuffer flip()` 方法

`ByteBuffer` 类提供了 `readerIndex()` 和 `writerIndex()` 方法。

`readerIndex()` 和 `writerIndex()` 方法返回 `0` 到 `writerIndex()` 之间的 `readerIndex` 和 `writerIndex` 的值。调用 `readerIndex()` 方法返回 `0`，调用 `writerIndex()` 方法返回 `discardReadBytes` 的值。调用 `compact()` 方法后，`readerIndex` 和 `writerIndex` 的值如图15-4所示。调用 `ByteBuffer` 的 `position` 和 `limit` 方法返回 `WriterIndex` 和 `capacity` 的值。调用 `ByteBuffer` 的 `limit` 和 `capacity` 方法返回 `0`。

调用 `readerIndex()` 和 `writerIndex()` 方法返回 `0` 到 `writerIndex()` 之间的 `readerIndex` 和 `writerIndex` 的值。调用 `flip()` 方法后，`position` 和 `limit` 的值如图15-5所示。

调用 `ByteBuffer` 的 `flip()` 方法

图15-4 调用 `ByteBuffer` 的 `compact()` 方法

调用 `N` 个 `ByteBuffer` 的 `flip()` 方法

图15-5 调用 `N` 个 `ByteBuffer` 的 `flip()` 方法

调用 `M` 个 `N` 个 `ByteBuffer` 的 `flip()` 方法


```
discardReadBytes(ByteBuf 15-7)
```

```
ByteBuffer[] ByteBuffer[]  
put() throws BufferOverflowException  
put(ByteBuffer) ByteBuffer  
ByteBuffer ByteBuffer ByteBuffer  
ByteBuffer
```

if

(this.

```
buffer.remaining() < needSize)
{
```

int

```
toBeExtSize = needSize < 128 ? needSize : 128;  
ByteBuffer tmpBuffer =  
ByteBuffer.allocate(this.buffer.capacity() + toBeExtSize);  
this.
```

```
buffer.flip();  
tmpBuffer.put(this.
```

```
buffer);  
this.
```

```
buffer = tmpBuffer;  
}
```


ByteBuffer.read()ByteBuffer.get()API15-1
图 15-1

图 15-1 ByteBuffer API

2 write

ByteBuffer.write()ByteBuffer.put()API15-2
图 15-2

图 15-2 ByteBuffer API

3 readerIndexwriterIndex

Netty readerIndexwriterIndexByteBuffer15-11
图 15-11

图 15-11 ByteBuffer readerIndexwriterIndex

ByteBuffer.read()readerIndex()readerIndex()writerIndex()capacity()0readerIndex()discardReadBytes()ByteBufferTCP TCP discardReadBytes()discardReadBytes()

4 Discardable bytes

Figure 4-1: Java's discardable bytes. The diagram shows a sequence of bytes in a buffer. The first N bytes are marked as "discardable". The next L bytes are marked as "readable". The remaining M bytes are marked as "unreadable".

Figure 4-2: Discardable bytes. The diagram shows a buffer of size R . The first M bytes are marked as "discardable". The next L bytes are marked as "readable". The remaining M bytes are marked as "unreadable". The buffer is represented as a `ByteBuffer` object. The `discardReadBytes` method is used to discard the first M bytes. The buffer is then represented as a `ByteBuffer` object with a new capacity of $R + 1$.

`ByteBuffer discardReadBytes(int n)`

Figure 4-3: Discardable bytes. The diagram shows a buffer of size R . The first M bytes are marked as "discardable". The next L bytes are marked as "readable". The remaining M bytes are marked as "unreadable".

Figure 4-3: `discardReadBytes` method. The diagram shows a buffer of size R . The first M bytes are marked as "discardable". The next L bytes are marked as "readable". The remaining M bytes are marked as "unreadable".

Figure 4-4: Discardable bytes. The diagram shows a buffer of size R . The first M bytes are marked as "discardable". The next L bytes are marked as "readable". The remaining M bytes are marked as "unreadable".

Figure 4-4: `discardReadBytes` method. The diagram shows a buffer of size R . The first M bytes are marked as "discardable". The next L bytes are marked as "readable". The remaining M bytes are marked as "unreadable".

Figure 4-5: Discardable bytes. The diagram shows a buffer of size R . The first M bytes are marked as "discardable". The next L bytes are marked as "readable". The remaining M bytes are marked as "unreadable". The buffer is represented as a `ByteBuffer` object. The `discardReadBytes` method is used to discard the first M bytes. The buffer is then represented as a `ByteBuffer` object with a new capacity of $R + 1$.

Figure 4-5: `discardReadBytes` method. The diagram shows a buffer of size R . The first M bytes are marked as "discardable". The next L bytes are marked as "readable". The remaining M bytes are marked as "unreadable".

Figure 4-6: Discardable bytes. The diagram shows a buffer of size R . The first M bytes are marked as "discardable". The next L bytes are marked as "readable". The remaining M bytes are marked as "unreadable". The buffer is represented as a `ByteBuffer` object. The `discardReadBytes` method is used to discard the first M bytes. The buffer is then represented as a `ByteBuffer` object with a new capacity of $R + 1$.

5 Readable bytesWritable bytes

read()skip()readerIndex()
readerIndex()IndexOutOfBoundsException()
ByteBuffer()readerIndex()0

write()writerIndex()
writerIndex()IndexOutOfBoundsException()ByteBuffer()
readerIndex()0ByteBuffer()writerIndex()ByteBuffer()

6 Clear

JDK ByteBufferclear()NUL
0x00position()limit()mark()ByteBuffer()
readerIndex()writerIndex()

Clear()15-15

15-15 clear

Clear() 15-16

15-16 clear

7 MarkRest

ByteBuffer 的 mark 方法，用于标记当前的位置，以便以后可以回到该位置。mark 方法返回一个布尔值，表示是否成功标记。如果成功，则返回 true；否则，返回 false。

在 JDK 的 ByteBuffer 中，mark 方法返回一个布尔值，表示是否成功标记。如果成功，则返回 true；否则，返回 false。rest 方法返回一个布尔值，表示是否成功重置。如果成功，则返回 true；否则，返回 false。

15-17 mark 方法

reset 方法

15-18 rest 方法

Netty 的 ByteBuffer 的 rest 方法，用于重置当前的位置，以便以后可以回到该位置。rest 方法返回一个布尔值，表示是否成功重置。如果成功，则返回 true；否则，返回 false。

- markReaderIndex 方法，用于标记当前的读者索引。该方法返回一个布尔值，表示是否成功标记。如果成功，则返回 true；否则，返回 false。
- resetReaderIndex 方法，用于重置当前的读者索引。该方法返回一个布尔值，表示是否成功重置。如果成功，则返回 true；否则，返回 false。
- markWriterIndex 方法，用于标记当前的写入索引。该方法返回一个布尔值，表示是否成功标记。如果成功，则返回 true；否则，返回 false。
- resetWriterIndex 方法，用于重置当前的写入索引。该方法返回一个布尔值，表示是否成功重置。如果成功，则返回 true；否则，返回 false。

15-19 方法

15-19 mark 和 rest 方法

8 个方法

ByteBuffer 的 `compareTo` 方法返回 `\r\n` 的字典序与 `NUL(0x00)` 的字典序之差。

ByteBuffer 的 `compareTo` 方法返回 `\r\n` 的字典序与 `NUL(0x00)` 的字典序之差。

1. `indexOf(int fromIndex, int toIndex, byte value)` 方法返回 ByteBuffer 中从 `fromIndex` 到 `toIndex` 的范围内，首次出现 `value` 的索引。如果未找到，则返回 `-1`。

2. `bytesBefore(byte value)` 方法返回 ByteBuffer 中在 `readerIndex` 之前，首次出现 `value` 的索引。如果未找到，则返回 `-1`。

3. `bytesBefore(int length, byte value)` 方法返回 ByteBuffer 中从 `readerIndex` 开始，长度为 `length` 的子序列中，首次出现 `value` 的索引。如果未找到，则返回 `-1`。如果 `length` 为 0，则返回 `readerIndex`。

4. `bytesBefore(int index, int length, byte value)` 方法返回 ByteBuffer 中从 `index` 开始，长度为 `length` 的子序列中，首次出现 `value` 的索引。如果未找到，则返回 `-1`。如果 `length` 为 0，则返回 `index`。

5. `forEachByte(ByteBufProcessor processor)` 方法遍历 ByteBuffer 中的所有字节，并对每个字节调用 `processor` 方法。如果 `processor` 返回 `true`，则返回 `1`。

6. `forEachByte(int index, int length, ByteBufProcessor processor)` 方法遍历 ByteBuffer 中从 `index` 开始，长度为 `length` 的子序列中的所有字节，并对每个字节调用 `processor` 方法。如果 `processor` 返回 `true`，则返回 `1`。

ByteBufProcessor[] processors = new ByteBufProcessor[writerIndex - readerIndex - 1];

```
for (int i = 0; i < processors.length; i++) {
    processors[i] = new ByteBufProcessor(
        buf, readerIndex + i, writerIndex - i - 1);
}
```

```
for (int i = 0; i < processors.length; i++) {
    processors[i].process(buf, readerIndex + i, writerIndex - i - 1);
}
```

Netty 4.0.0-RC1 中，ByteBufProcessor 接口定义如下：

1. FIND_NUL (0x00)

2. FIND_CR ('\r')

3. FIND_LF ('\n')

4. FIND_CRLF ('\r') + '\n')

5. FIND_LINEAR_WHITESPACE (' ' + '\t')

15-20 个字节

15-20 个字节

9. Derived buffers

❶ ByteBuffer.nioBuffer() 返回 ByteBuffer 的只读视图。ByteBuffer 的只读视图与原始的 ByteBuffer 共享底层数据。ByteBuffer 的只读视图与原始的 ByteBuffer 具有相同的容量和位置。ByteBuffer 的只读视图与原始的 ByteBuffer 具有相同的顺序。ByteBuffer 的只读视图与原始的 ByteBuffer 具有相同的标志。

❷ ByteBuffer.nioBuffer(int index, int length) 返回 ByteBuffer 的只读视图。ByteBuffer 的只读视图与原始的 ByteBuffer 共享底层数据。ByteBuffer 的只读视图与原始的 ByteBuffer 具有相同的容量和位置。ByteBuffer 的只读视图与原始的 ByteBuffer 具有相同的顺序。ByteBuffer 的只读视图与原始的 ByteBuffer 具有相同的标志。

11 ByteBuffer 的 set 和 get

ByteBuffer 的 set 和 get 方法用于设置和获取字节。ByteBuffer 的 set 方法用于设置字节。ByteBuffer 的 get 方法用于获取字节。

ByteBuffer API 15-21

15-21 ByteBuffer API

ByteBuffer API 15-22

15-22 ByteBuffer API

ByteBuffer 的 get 和 set 方法用于设置和获取字节。ByteBuffer 的 get 方法用于获取字节。ByteBuffer 的 set 方法用于设置字节。ByteBuffer 的 write 方法用于写入字节。ByteBuffer 的 read 方法用于读取字节。ByteBuffer 的 15-23

15-23 ByteBuffer API

15.2 ByteBuffer

ByteBuffer 是 Java 7 引入的一个类，它提供了对字节缓冲区的操作。ByteBuffer 是 java.nio 包下的一个类，它提供了对字节缓冲区的操作。ByteBuffer 是 java.nio 包下的一个类，它提供了对字节缓冲区的操作。

15.2.1 ByteBuffer 的用途

ByteBuffer 的用途非常广泛，它主要用于处理字节数据。ByteBuffer 的用途非常广泛，它主要用于处理字节数据。

图 15-24 ByteBuffer 的用途

ByteBuffer 的用途非常广泛，它主要用于处理字节数据。

ByteBuffer 的用途非常广泛，它主要用于处理字节数据。ByteBuffer 的用途非常广泛，它主要用于处理字节数据。ByteBuffer 的用途非常广泛，它主要用于处理字节数据。

ByteBuffer 的用途非常广泛，它主要用于处理字节数据。ByteBuffer 的用途非常广泛，它主要用于处理字节数据。ByteBuffer 的用途非常广泛，它主要用于处理字节数据。

ByteBuffer 的用途非常广泛，它主要用于处理字节数据。ByteBuffer 的用途非常广泛，它主要用于处理字节数据。ByteBuffer 的用途非常广泛，它主要用于处理字节数据。

ByteBuffer 的用途非常广泛，它主要用于处理字节数据。ByteBuffer 的用途非常广泛，它主要用于处理字节数据。ByteBuffer 的用途非常广泛，它主要用于处理字节数据。

ByteBuffer 的包装类，Netty 的 ByteBuffer 包装类

ByteBuffer 的包装类，Netty 的 ByteBuffer 包装类

15.2.2 AbstractByteBuffer

AbstractByteBuffer 是 ByteBuffer 的子类，AbstractByteBuffer 是 ByteBuffer 的子类

1. 构造函数

mark 方法，15-25

15-25 AbstractByteBuffer

leakDetector 是 static 的，ResourceLeakDetector 是 ResourceLeakDetector 的子类

AbstractByteBuffer 是 ByteBuffer 的子类，DirectByteBuffer 是 AbstractByteBuffer 的子类

2. 方法

ByteBuffer 的 UnpooledHeapByteBuffer 和 UnpooledDirectByteBuffer 是 ByteBuffer 的子类

ByteBuffer

ByteBuffer.read() 方法返回一个 ByteBuffer 对象，该对象包含从源缓冲区读取的字节。
ByteBuffer

read() 方法 15-26

15-26 ByteBuffer.read()

ByteBuffer.readBytes(byte[] dst, int dstIndex, int length) 方法从源缓冲区读取字节到目标缓冲区。
15-27

15-27 readBytes(byte[] dst, int dstIndex, int length)

ByteBuffer.readBytes() 方法 15-28

15-28 readBytes(byte[] dst, int dstIndex, int length)

ByteBuffer.read() 方法返回一个 ByteBuffer 对象，该对象包含从源缓冲区读取的字节。
ByteBuffer.readBytes() 方法从源缓冲区读取字节到目标缓冲区。
ByteBuffer.readBytes() 方法从源缓冲区读取字节到目标缓冲区。

ByteBuffer.getBytes() 方法从源缓冲区读取字节到目标缓冲区。
15-29

15-29 ByteBuffer.getBytes()

ByteBuffer.readerIndex += length 方法从源缓冲区读取字节到目标缓冲区。
ByteBuffer.readerIndex += length

3

ByteBuffer API 15-30

15-30

writeBytes(byte[] src, int srcIndex, int length) 0 srcIndex srcIndex + length ByteBuffer 15-31

15-31

15-32

15-32

0 IllegalArgumentException 0 ByteBuffer IndexOutOfBoundsException

ByteBuffer JDK ByteBuffer

JDK ByteBuffer POJO

Netty 的 ByteBuffer 的容量是 4M，当容量不足时，会调用 calculateNewCapacity 方法，将容量扩大为原来的 1.5 倍，即 writerIndex + minWritableBytes * 1.5。

Netty 的 ByteBuffer 的容量是 4M，当容量不足时，会调用 calculateNewCapacity 方法，将容量扩大为原来的 1.5 倍，即 writerIndex + minWritableBytes * 1.5。

图 15-33 容量扩容示意图

Netty 的 ByteBuffer 的容量是 4M，当容量不足时，会调用 calculateNewCapacity 方法，将容量扩大为原来的 1.5 倍，即 writerIndex + minWritableBytes * 1.5。

Netty 的 ByteBuffer 的容量是 4M，当容量不足时，会调用 calculateNewCapacity 方法，将容量扩大为原来的 1.5 倍，即 writerIndex + minWritableBytes * 1.5。

Netty 的 ByteBuffer 的容量是 4M，当容量不足时，会调用 calculateNewCapacity 方法，将容量扩大为原来的 1.5 倍，即 writerIndex + minWritableBytes * 1.5。

Netty 的 ByteBuffer 的容量是 4M，当容量不足时，会调用 calculateNewCapacity 方法，将容量扩大为原来的 1.5 倍，即 writerIndex + minWritableBytes * 1.5。

Netty 的 ByteBuffer 的容量是 4M，当容量不足时，会调用 calculateNewCapacity 方法，将容量扩大为原来的 1.5 倍，即 writerIndex + minWritableBytes * 1.5。

ByteBuffer.mark() 方法用于在 ByteBuffer 中设置一个标记，以便在需要时能够快速定位到该位置。该方法的返回值为 ByteBuffer 对象本身。图 15-34 展示了该方法的调用。

图 15-34 ByteBuffer.mark() 方法

4. 读取数据

ByteBuffer 提供了多种方法来读取数据。图 15-35 展示了使用 mark() 和 rest() 方法读取数据的示例。

图 15-35 使用 mark() 和 rest() 方法读取数据

ByteBuffer 还提供了多种方法来写入数据。图 15-36 展示了使用 put() 方法写入数据的示例。

图 15-36 使用 put() 方法写入数据

ByteBuffer 还提供了多种方法来检查数据的有效性。图 15-37 展示了使用 isReadOnly() 方法检查数据是否为只读的示例。如果返回值为 true，则表示该 ByteBuffer 是只读的，任何尝试修改其内容的操作都会抛出 IndexOutOfBoundsException 异常。图 15-38 展示了使用 isReadOnly() 方法检查数据是否为只读的示例。

5. 其他方法

ByteBuffer 还提供了多种方法来丢弃数据。图 15-39 展示了使用 discardReadBytes() 方法丢弃已读取的数据的示例。该方法接受一个参数，表示要丢弃的字节数。图 15-40 展示了使用 discardSomeReadBytes() 方法丢弃部分已读取的数据的示例。该方法接受一个参数，表示要丢弃的字节数。图 15-41 展示了使用 discardReadBytes() 方法丢弃已读取的数据的示例。

图 15-37 使用 isReadOnly() 方法检查数据是否为只读

ByteBuffer 还提供了多种方法来重置数据。图 15-42 展示了使用 reset() 方法重置数据到初始位置的示例。该方法会将 ByteBuffer 的 position 属性重置为 0，并清除所有已写入的数据。图 15-43 展示了使用 rewind() 方法重置数据到初始位置的示例。该方法会将 ByteBuffer 的 position 属性重置为 0，并清除所有已写入的数据。

15-40

15-40

IndexOutOfBoundsException
IllegalArgumentException

IndexOutOfBoundsException
length

15.2.3 AbstractReferenceCountedByteBuf

JVM

1

AbstractReferenceCountedByteBuf 15-41

15-41 AbstractReferenceCountedByteBuf

refCntUpdater
AtomicIntegerFieldUpdater
REFCNT_
FIELD_OFFSET
refCnt
AbstractReferenceCountedByteBuf
JDK
SUN
JDK
sun.misc.Unsafe
objectFieldOffset

ByteBuffer 的 UnpooledUnsafeDirectByteBuffer 和 PooledUnsafeDirectByteBuffer

volatile refCnt volatile
volatile

2

retain
15-42

15-42 retain

1
ByteBuffer
0
IllegalReferenceCountException
IllegalReferenceCountException
compareAndSet
compareAndSet
CAS
Java CAS

15-43

15-43

retain
refCnt
== 1
deallocate
ByteBuffer

15.2.4 UnpooledHeapByteBuffer

UnpooledHeapByteBuffer 是堆内存的 ByteBuffer，它是最简单的 ByteBuffer 实现，它直接操作堆内存，不需要通过 PooledByteBuffer 的池化操作。它适用于 I/O 操作，但不适用于需要频繁创建和销毁的场景。UnpooledHeapByteBuffer 的创建和销毁成本较低，但它的内存管理不如 PooledByteBuffer 高效。

UnpooledHeapByteBuffer 是 PooledByteBuffer 的子类，它继承了 PooledByteBuffer 的所有方法。UnpooledHeapByteBuffer 的创建和销毁成本较低，但它的内存管理不如 PooledByteBuffer 高效。UnpooledHeapByteBuffer 的创建和销毁成本较低，但它的内存管理不如 PooledByteBuffer 高效。

UnpooledHeapByteBuffer 的创建和销毁成本较低，但它的内存管理不如 PooledByteBuffer 高效。

1. 创建和销毁

UnpooledHeapByteBuffer 的创建和销毁成本较低，但它的内存管理不如 PooledByteBuffer 高效。

15-44 UnpooledHeapByteBuffer 的创建和销毁

ByteBufferAllocator 是 ByteBuffer 的工厂类，它负责创建和销毁 ByteBuffer。UnpooledHeapByteBuffer 是 ByteBufferAllocator 的子类，它继承了 ByteBufferAllocator 的所有方法。ByteBufferAllocator 的创建和销毁成本较低，但它的内存管理不如 PooledByteBuffer 高效。

JDK 的 ByteBuffer 是 byte 的包装类，它负责创建和销毁 ByteBuffer。JDK 的 ByteBuffer 的创建和销毁成本较低，但它的内存管理不如 PooledByteBuffer 高效。JDK 的 ByteBuffer 的创建和销毁成本较低，但它的内存管理不如 PooledByteBuffer 高效。

15-45 JDK ByteBuffer 的创建和销毁

2. 内存管理

ByteBuffer extends AbstractByteBuffer implements ByteBuffer, CompactByteBuffer, PooledByteBuffer, UnpooledHeapByteBuffer
15-46

15-46 ByteBuffer

ByteBuffer.allocate(0) throws IllegalArgumentException

ByteBuffer.allocate(byte[] newArray) = new byte[newCapacity] System.arraycopy(array, 0, newArray, 0, array.length) setArray 15-47

15-47 ByteBuffer

ByteBuffer.allocate(tmpNioBuf)

ByteBuffer.allocate(System.arraycopy(array, readerIndex, newArray, readerIndex, writerIndex - readerIndex))

System.arraycopy(array, readerIndex, newArray, readerIndex, writerIndex - readerIndex);

ByteBuffer.setArray(array)

3

```

    setBytes(int index, byte[] src, int srcIndex,
int length)UnpooledHeapByteBuffer
15-48

```

□15-48 □□□□□□

15-49

□15-49 □□□□□□□□

```

    index length 0
IllegalArgumentException
IndexOutOfBoundsException srcIndex srcCapacity
indexSystem.arraycopy(src, srcIndex,
array, index, length)

```

ByteBuf.set/get

4.4.4.4 JDK ByteBuffer

```

    00JDK NIO ByteBuffer0000000000000000ByteBuf00byte
    00000NIO0ByteBuffer000wrap000000byte00000ByteBuffer0
    00JDK000000000015-50000

```

□15-50 JDK ByteBuffer□warp□□□□

```

0000000000000000UnpooledHeapByteBuffer15-51
000

```

15-51 UnpooledHeapByteBuffer.wrap

ByteBuffer.slice() slice() nioBuffer() ByteBuffer slice()

5

ByteBuffer() isDirect() false 15-52

- isDirect() false 15-52

15-52 UnpooledHeapByteBuffer.isDirect

- hasArray() UnpooledHeapByteBuffer() true
- array() UnpooledHeapByteBuffer() 15-53

15-53 UnpooledHeapByteBuffer.array

array() hasArray() false ByteBuffer() array()

- arrayOffset() hasMemoryAddress() memoryAddress() 15-54

15-54 UnpooledHeapByteBuffer.address

UnsafeByteBuf 是 SUN JDK 的
sun.misc.Unsafe 类的一个子类。sun.misc.Unsafe 类是
sun.misc.Unsafe 类的一个子类。OPEN JDK 的
DOC 文档。

UnpooledDirectByteBuf 和 UnpooledHeapByteBuf 是
java.nio.DirectByteBuffer 和
UnpooledHeapByteBuf 的子类。UnpooledDirectByteBuf 和
UnpooledDirectByteBuf 是

15.2.5 PooledByteBuf 类

ByteBuf 类是
类。

1 PoolArena

Arena 是 Memory Arena 的一个子类。
PoolArena 是 Netty 的

是 Memory Arena 的
是 Memory Arena 的
是 Memory Arena 的
是 Memory Arena 的

Memory Arena 是 Netty 的 PoolArena 的 Chunk
Chunk 是 Page 的
Chunk 是 Page 的 PoolArena 的 Chunk 是 15-55

2 PoolChunk

Chunk由若干个Page组成，Netty的Chunk由Page组成，Chunk由16个Page组成，Page由15-56字节组成。

图15-56 Chunk

Page由4个Chunk组成，Chunk由64字节(4×16)组成，5个1个Page组成，4个Page组成2个Page。

Memory Arena由若干个Chunk组成，Chunk由16个Page组成，3个4个Page组成，Memroy Arena由16个Page组成，3个Page组成。

Page由4个Chunk组成，Chunk由64字节(4×16)组成，5个1个Page组成，4个Page组成2个Page。

3 PoolSubpage

Page由Netty的Page组成，Page由8个Page组成，4个Page组成2个Page，8个Page组成1个Page。

Page 4 Page 1 Page 2 Page

```

Page0longlong
014PagePage1
longlong4
128PagePage1
long2128

```

15-57

□15-57 PoolSubpage□□□□□

4

```

    ChunkPage
    Chunk
    Page

```

API

15.2.6 PooledDirectByteBuffer

```
PooledDirectByteBufferUnPooledDirectByteBuffer
```

1

```
        PooledDirectByteBuffer newBuf =
        15-58
```

15-58 PooledDirectByteBuffer

```
        Recycler<PooledDirectByteBuffer>
        PooledDirectByteBuffer 1
```

2

```
        PooledDirectByteBuffer
        copy(int index, int length)
        15-59
```

15-59 PooledDirectByteBuffer copy

```
        PooledByteBufferAllocator
        ByteBuffer PooledByteBufferAllocator directBuffer
        AbstractByteBufferAllocator directBuffer
        15-60
```

15-60 AbstractByteBufferAllocator

```
        newDirectBuffer
        ByteBuffer ByteBuffer
        15-61 15-62
```

15-61

15-62

ByteBuffer 的池化操作
图 15-63

3. 池化操作

UnpooledHeapByteBuffer 和 PooledDirectByteBuffer 的池化操作
图 15-63

图 15-63 PooledDirectByteBuffer 的池化操作

ByteBuffer 的池化操作
图 15-63

15.3 ByteBuffer 的池化操作

ByteBuffer 的池化操作

15.3.1 ByteBufferHolder

ByteBufferHolder 是 ByteBuffer 的池化操作，Netty 使用 ByteBufferHolder 来管理 HTTP 请求和响应。ByteBufferHolder 是 NIO ByteBuffer 的池化操作，Netty 使用 ByteBufferHolder 来管理 HTTP 请求和响应。ByteBufferHolder 是 NIO ByteBuffer 的池化操作，Netty 使用 ByteBufferHolder 来管理 HTTP 请求和响应。

ByteBufferHolder 是 Netty 使用 ByteBufferHolder 来管理 HTTP 请求和响应。ByteBufferHolder 是 Netty 使用 ByteBufferHolder 来管理 HTTP 请求和响应。ByteBufferHolder 是 Netty 使用 ByteBufferHolder 来管理 HTTP 请求和响应。

图 15-64 ByteBufferHolder 的池化操作

15.3.2 ByteBufAllocator

ByteBufAllocator 是 Netty 中用于分配和释放内存的接口。它定义了分配和释放内存的方法，以及管理内存池的方法。图 15-65 展示了 ByteBufAllocator 的接口定义。

图 15-65 ByteBufAllocator 的接口定义

图 15-3 展示了 ByteBufAllocator 的 API 接口。

图 15-3 ByteBufAllocator 的 API 接口

15.3.3 CompositeByteBuf

CompositeByteBuf 是 Netty 中用于管理多个 ByteBuf 的接口。它定义了管理多个 ByteBuf 的方法，以及管理内存池的方法。图 15-66 展示了 CompositeByteBuf 的接口定义。

CompositeByteBuf 是 Netty 中用于管理多个 ByteBuf 的接口。它定义了管理多个 ByteBuf 的方法，以及管理内存池的方法。图 15-66 展示了 CompositeByteBuf 的接口定义。

图 15-66 展示了 CompositeByteBuf 的 API 接口。

图 15-66 CompositeByteBuf 的 API 接口

图 15-66 展示了 CompositeByteBuf 的 API 接口。

图 15-66 CompositeByteBuf 的 API 接口

图15-66 CompositeByteBuffer

ByteBufferComponent和ByteBufferComponentByteBuffer
ByteBufferByteBuffer15-67

图15-67 Component

CompositeByteBufferByteBuffer15-68

图15-68 CompositeByteBufferByteBuffer

ByteBuffer15-69

图15-69 CompositeByteBufferByteBuffer

ByteBufferComponent

15.3.4 ByteBufferUtil

ByteBufferUtilByteBuffer
15-70

图15-70 ByteBufferUtil

1 encodeString(ByteBufAllocator alloc, CharBuffer src,
Charset charset) src charset
ByteBufAllocatorByteBuffer

2 decodeString(ByteBuffer src, Charset charset) 将 ByteBuffer charset 的 ByteBuffer 解码成字符串

hexDump(ByteBuf) 将 ByteBuf 以十六进制形式打印到控制台

hexDump(15-71) 将 15-71 的十六进制数据打印到控制台

15-71 hexDump

15.4 总结

ByteBuffer API 与 Netty 的 ByteBuffer 区别
ByteBuffer 与 Netty 的 ByteBuffer 区别
ByteBuffer 与 Netty 的 ByteBuffer 区别

ByteBuffer 与 Netty 的 ByteBuffer 区别
ByteBuffer 与 Netty 的 ByteBuffer 区别
ByteBuffer 与 Netty 的 ByteBuffer 区别

ByteBuffer 与 Netty 的 ByteBuffer 区别

16 ChannelUnsafe

Channel——JDK NIO
java.nio.SocketChannel java.nio.ServerSocketChannel
I/O

IOChannelNettyChannelI/O

```
UnsafeChannel
Channel
Channel
Netty
Unsafe
Channel
```

--	--	--	--	--	--	--	--	--

- Channel [][][]
- Unsafe[][][]
- Channel[[] [] [] [] [] [] [] []]
- Unsafe[[] [] [] [] [] [] [] []]

16.1 Channel

```
io.netty.channel.Channel
Netty
Channel
EventLoop
ByteBufAllocator
pipeline
```

Channel API

16.1.1 Channel

Channel 是 Netty 中负责 I/O 操作的类，JDK NIO 中 Channel 是负责 I/O 操作的类。

1. JDK 中 SocketChannel 和 ServerSocketChannel 是 Channel 的子类。

2. JDK 中 SocketChannel 和 ServerSocketChannel 是 I/O 操作的 SPI 接口，SPI 接口是 I/O 操作的 SPI 接口，ServerSocketChannel 和 SocketChannel 是 Channel 的子类。

3. Netty 中 Channel 是 Netty 中负责 I/O 操作的 ChannelPipeline 接口，TCP 是 JDK 中 SocketChannel 和 ServerSocketChannel 的子类。

4. Channel 是 Channel 的子类。

4. Netty 中 Channel 是 Netty 中负责 I/O 操作的 ChannelPipeline 接口，TCP 是 JDK 中 SocketChannel 和 ServerSocketChannel 的子类。

1. Channel 是 Facade 接口，I/O 是 I/O 接口。

2. Channel 是 SocketChannel 和 ServerSocketChannel 的子类。

3. Channel Channel
Channel

16.1.2 Channel

Channel

1. I/O

Channel I/O 16-1

16-1 Channel I/O API

API API

1. Channel read() Channel inbound
ChannelHandler.channelRead(ChannelHandlerContext,
Object) API
ChannelHandler.channelReadComplete (Channel
HandlerContext) ChannelHandler
flush Channel

2. ChannelFuture write(Object msg) msg
ChannelPipeline Channel write
flush Channel

3. ChannelFuture write(Object msg, ChannelPromise
promise) write(Object msg) ChannelPromise

ConnectException

ChannelHandler.connect(ChannelHandlerContext,
SocketAddress, SocketAddress, ChannelPromise)

10 ChannelFuture connect(SocketAddress
remoteAddress, SocketAddress localAddress) 9
 localAddress

11 ChannelFuture connect(SocketAddress
remoteAddress, ChannelPromise promise) 9
 ChannelPromise

12 connect(SocketAddress remoteAddress,
SocketAddress localAddress, ChannelPromise promise)
11

13 ChannelFuture bind(SocketAddress localAddress)
 Socket localAddress
ChannelHandler.bind(ChannelHandlerContext,
SocketAddress, ChannelPromise)

14 ChannelFuture bind(SocketAddress localAddress,
ChannelPromise promise) 13
ChannelPromise

15 ChannelConfig config() Channel
CONNECT_TIMEOUT_MILLIS

16 boolean isOpen() Channel

17 boolean isRegistered() Channel EventLoop

18 boolean isActive() Channel

19 ChannelMetadata metadata() Channel TCP

20 SocketAddress localAddress() Channel

21 SocketAddress remoteAddress() Channel Socket

2 API

eventLoop() Channel EventLoop I/O eventLoop() Channel EventLoop EventLoop Reactor Netty NioTask

metadata() TCP Socket TCP TCP TCP Netty Channel TCP Channel ChannelMetadata TCP metadata() Channel TCP

parent() Channel Channel Channel Channel Channel ServerSocketChannel

Channel id() ChannelId ChannelId
Channel

1 MAC EUI-48 EUI-64

2 ID

3 — System.currentTimeMillis()

4 — System.nanoTime()

5 32

6 32

16.2 Channel

Channel —
Channel-io.netty.channel.socket.nio.NioServerSocketChannel
io.netty.channel.socket.nio. NioSocketChannel
Channel

16.2.1 Channel

NioSocketChannel
NioServerSocketChannel

NioServerSocketChannel 16-2

16-2 NioServerSocketChannel

□□□NioSocketChannel□□□□□□□□16-3□□□

□16-3 NioSocketChannel □□□□□□

16.2.2 AbstractChannel□□□□

1□□□□□□□

□□□AbstractChannel□□□□□□□□□□□□□□□□16-4□□□

□16-4 AbstractChannel□□□□□□

□□□□□□□□□□□□

- CLOSED_CHANNEL_EXCEPTION□□□□□□□□□□□□
- NOT_YET_CONNECTED_EXCEPTION□□□□□□□□□□□□

□□□□□□□□□□□□□□□□□□□□□□StackTraceElement□

estimatorHandle□□□□□□□□□□□□□□□□□□□□□□□□□□□□

□□□□Channel□□□□□□□□AbstractChannel□□□□□□□□□□□□
□□□□□□□□□□□□□□□□□□□□□□□□□□□□

- parent□□□□Channel□
- id□□□□□□□□□□□□□□ID□
- unsafe□Unsafe□□□
- pipeline□□□Channel□□□DefaultChannelPipeline□
- eventLoop□□□Channel□□□EventLoop□

.....

AbstractChannel 是 Channel 的
父接口。AbstractChannel 定义了 Channel 的
基本 API。

2. API

ChannelPipeline 是 I/O Channel 的
Netty 实现。ChannelPipeline 是 I/O
Channel 的 ChannelHandler 的
集合。ChannelPipeline 是 AOP 的
实现。

I/O Channel 的 DefaultChannelPipeline 是
DefaultChannelPipeline 的 ChannelHandler
16-5

16-5 AbstractChannel 的 I/O 实现

AbstractChannel 的 API 定义了 localAddress() 和
remoteAddress() 16-6

16-6 AbstractChannel remoteAddress

unsafe 的 remoteAddress
Channel

16.2.3 AbstractNioChannel

1. 实现

16-7

16-7 AbstractNioChannel

NIO Channel NioSocketChannel
NioServerSocketChannel
java.nio.SocketChannel
java.nio.ServerSocketChannel
SelectableChannel
SelectableChannel I/O

readInterestOp JDK SelectionKey OP_READ

volatile SelectionKey SelectionKey Channel
EventLoop Channel
SelectionKey SelectionKey
volatile

ChannelPromise
ScheduledFuture

2 API

AbstractNioChannel API Channel
16-8

16-8 AbstractNioChannel

selected Selectable
Channel register Channel EventLoop
Selectable Channel 16-9

16-9 JDK SelectableChannel 的常量

Channel 的常量在 Channel 的父类 SelectableChannel 中定义。

- public static final int OP_READ = 1 << 0
- public static final int OP_WRITE = 1 << 2
- public static final int OP_CONNECT = 1 << 3
- public static final int OP_ACCEPT = 1 << 4

AbstractNioChannel 的常量在 AbstractNioChannel 的父类 NioChannel 中定义。SelectionKey 的常量在 SelectionKey 的父类 AbstractNioChannel 中定义。selectionKey 的常量在 selectionKey 的父类 NioChannel 中定义。

selectionKey 的常量在 selectionKey 的父类 NioChannel 中定义。CancelledKeyException 的常量在 CancelledKeyException 的父类 IOException 中定义。selectNow() 的常量在 selectNow() 的父类 SelectionKey 中定义。selected 的常量在 selected 的父类 SelectionKey 中定义。CancelledKeyException 的常量在 CancelledKeyException 的父类 IOException 中定义。selectionKey 的常量在 selectionKey 的父类 NioChannel 中定义。JDK API 的常量在 JDK API 的父类 NIO 中定义。BUG 的常量在 CancelledKeyException 的父类 IOException 中定义。

16-10
16-10

16-10

Channel SelectionKey
Channel SelectionKey
0 interestOps |
readInterestOp selectionKey interestOps
16-10

JDK NIO SelectionKey
16-11

16-11 SelectionKey

16.2.4 AbstractNioByteChannel

Runnable flushTask
16-11

doWrite ChannelOutboundBuffer in
16-12

16-12 doWrite ChannelOutboundBuffer in 1

ChannelOutboundBuffer
clearOpWrite 16-13

图16-13 伪代码

```
if (SelectionKey.isWritable()) {
    SelectionKey.OP_WRITE
    // 0 表示 SelectionKey.isWritable() 返回 true
    // SelectionKey.OP_WRITE 表示 SelectionKey
    SelectionKey
}
```

图16-14 伪代码

图16-14 doWrite(ChannelOutboundBuffer in) 伪代码2

```
ByteBuffer buf = in.getBuffer();
ByteBuffer buf = in.getBuffer();
// 0 表示 buf 为空
// 0 表示 buf 为空
// 0 表示 buf 为空
```

图16-15 伪代码

```
ByteBuffer buf = in.getBuffer();
// -1 表示 Channel 已关闭
// Channel
// I/O 操作
// I/O 操作
// I/O 操作
// NioTask
// I/O 操作
```

图16-15 伪代码

图16-15 doWrite(ChannelOutboundBuffer in) 伪代码3

```
doWriteBytes(Channel)
// 0 表示 TCP 窗口大小为 ZERO_WINDOW
```

0 CPU I/O I/O
setOpWrite true I/O

0
done true

ChannelOutboundBuffer
incompleteWrite
16-16

16-16 doWrite ChannelOutboundBuffer in 4

incompleteWrite 16-17

16-17

setOpWrite 16-18

16-18

SelectionKey
SelectionKey.OP_WRITE

SelectionKey.OP_WRITE Channel
SelectionKey.OP_WRITE
OP_WRITE Runnable

OP_WRITE Runnable
EventLoop Runnable flush()

ByteBuffer

ByteBuffer
ByteBuffer

16.2.5 AbstractNioMessageChannel

AbstractNioMessageChannel
ByteBuffer

doWrite(ChannelOutboundBuffer in)
16-19

16-19 AbstractNioMessageChannel

ChannelOutboundBuffer
ByteBuffer

AbstractNioByteChannel writeSpinCount
doWriteMessage(Object msg,
ChannelOutboundBuffer in) done
true writeSpinCount

SelectionKey.OP_WRITE
Channel

AbstractNioMessageChannel
AbstractNioByteChannel
ByteBuffer FileRegion POJO

16.2.6 AbstractNioMessageServerChannel

AbstractNioMessageServerChannel 是 NioSocketChannel 的父类，它定义了 EventLoopGroup 和 childGroup 等属性。NioSocketChannel 实现了 EventLoop 接口。16-20

16-20 AbstractNioMessageServerChannel

AbstractNioMessageServerChannel 是 NioSocketChannel 的父类，它定义了 childEventLoopGroup 和 EventLoopGroup 等属性。NioSocketChannel 实现了 Reactor 接口。16-21

16-21 childEventLoopGroup

16.2.7 NioServerSocketChannel

NioServerSocketChannel 是 NioSocketChannel 的子类，它实现了 API 接口。16-22

16-22 NioServerSocketChannel

NioServerSocketChannel 是 ChannelMetadata 的子类，它实现了 ServerSocketChannelConfig 接口。ServerSocketChannel 是 TCP 的。newSocket 方法返回 ServerSocketChannel，open 方法返回 ServerSocketChannel。

ServerSocketChannel 是 isActive 的。remoteAddress 是 javaChannel 的 doBind 方法。16-23

图16-23 NioServerSocketChannel的构造函数

图16-23展示了`java.net.ServerSocket`的`isBound`方法。该方法返回一个布尔值，表示该`ServerSocket`是否已经绑定到指定的`remoteAddress`。如果已经绑定，则返回`true`；否则返回`false`。该方法的实现依赖于`java.nio.ServerSocketChannel`的`isBound`方法。该方法的实现依赖于`backlog`参数。该方法的实现依赖于`API`。图16-24展示了

图16-24 ServerSocket的构造函数

图16-24展示了`Channel`的`doReadMessages(List<Object> buf)`方法。该方法返回一个布尔值，表示是否已经读取了消息。图16-25展示了

图16-25 NioServerSocketChannel doReadMessages方法

图16-25展示了`ServerSocketChannel`的`accept`方法。该方法返回一个`SocketChannel`对象。该方法的实现依赖于`NioServerSocketChannel`的`accept`方法。该方法的实现依赖于`EventLoop`和`SocketChannel`。该方法的实现依赖于`NioSocketChannel`。该方法的实现依赖于`List<Object> buf`。该方法的实现依赖于`1`。该方法的实现依赖于

图16-26展示了`NioServerSocketChannel`的`doReadMessages`方法。该方法的实现依赖于`NioSocketChannel`。该方法的实现依赖于

图16-26展示了`Channel`的`doReadMessages`方法。该方法的实现依赖于`Channel`。该方法的实现依赖于`Channel`。该方法的实现依赖于`UnsupportedOperationException`。该方法的实现依赖于`Exception`。图16-26展示了

图16-26 NioServerSocketChannel 的doReadMessages方法

16.2.8 NioSocketChannel

1

API 16-27

16-27 NioSocketChannel

Socket
java.nio.channels.SocketChannel.socket().bind()
java.nio.channels.SocketChannel.connect(SocketAddress remote) TCP
TCP

1 true

2 ACK false

3 I/O

2 NioSocketChannel selectionKey
OP_CONNECT I/O TCP
REST 16-28

16-28

2

16-29

16-29 NioSocketChannel 1

ByteBuffer 的 write 方法，调用的是 AbstractNioByteChannel 的 doWrite 方法。

doWrite 方法首先会检查 nioBufferCnt 是否大于 0，如果是，则调用 ChannelOutboundBuffer 的 write 方法，将 ByteBuffer 写入到 NioSocketChannel 中。NioSocketChannel 的 write 方法会调用 NIO SocketChannel 的 write 方法，将 ByteBuffer 写入到 SocketChannel 中。最后，doWrite 方法会返回 true，表示写入成功。如果 nioBufferCnt 不大于 0，则 doWrite 方法会返回 false，表示写入失败。

图 16-30 NioSocketChannel 的 doWrite 方法

图 16-31 NioSocketChannel 的 write 方法

图 16-31 NioSocketChannel 的 write 方法

Selector 是 NIO 中用于多路复用的核心组件。它负责监听多个通道上的事件，并将事件分发给相应的处理器。Selector 的 write 方法会调用 ChannelOutboundBuffer 的 write 方法，将 ByteBuffer 写入到 NioSocketChannel 中。NioSocketChannel 的 write 方法会调用 NIO SocketChannel 的 write 方法，将 ByteBuffer 写入到 SocketChannel 中。最后，Selector 的 write 方法会返回 true，表示写入成功。如果 Selector 的 write 方法返回 false，则表示写入失败。

NIO SocketChannel 的 write 方法，调用的是 SocketChannel 的 write 方法。SocketChannel 的 write 方法会调用 Socket 的 write 方法，将 ByteBuffer 写入到 Socket 中。最后，SocketChannel 的 write 方法会返回 true，表示写入成功。如果 SocketChannel 的 write 方法返回 false，则表示写入失败。

Selector 的 write 方法，调用的是 ChannelOutboundBuffer 的 write 方法。ChannelOutboundBuffer 的 write 方法会调用 NioSocketChannel 的 write 方法，将 ByteBuffer 写入到 NioSocketChannel 中。NioSocketChannel 的 write 方法会调用 NIO SocketChannel 的 write 方法，将 ByteBuffer 写入到 SocketChannel 中。最后，Selector 的 write 方法会返回 true，表示写入成功。如果 Selector 的 write 方法返回 false，则表示写入失败。

图 16-32 NioSocketChannel 的 write 方法

ByteBuffer + 16-33
true done
16-33

16-33 NioSocketChannel 5

ByteBuffer “”
Netty “” 16-34

16-34 NioSocketChannel 6

- 1 ChannelOutboundBuffer ByteBuffer
ByteBuffer
- 2 ByteBuffer ChannelOutboundBuffer
ByteBuffer
- 3 “” + ChannelOutboundBuffer
- 4

SocketChannel OP_WRITE
SocketChannel

3

NioSocketChannel NIO SocketChannel
Netty ByteBuf SocketChannel 16-35

16-35 NioSocketChannel

- java.nio.channels.SocketChannel JDK NIO SocketChannel
- length ByteBuf

SocketChannel L ByteBuf L ByteBuf
ByteBuf writeBytes 16-36

16-36 SocketChannel ByteBuf

setBytes(int index, ScatteringByteChannel in, int length) UnpooledHeapByteBuf 16-37

16-37 UnpooledHeapByteBuf set

SocketChannel java.nio.ByteBuffer
position writeIndex limit writeIndex+length JDK ByteBuffer
DOC 16-38

图16-38 java.nio.ByteBuffer的read方法

16.3 Unsafe类

Unsafe类是Channel类实现非阻塞I/O操作的核心API类。

图16-1 Unsafe API类

16.4 Unsafe类

Unsafe类是Channel类实现非阻塞I/O操作的核心API类。

16.4.1 Unsafe类

图16-39 Unsafe类

图16-39 Unsafe类

16.4.2 AbstractUnsafe类

1. register方法

register方法用于注册Unsafe类。Channel类实现EventLoop接口，DefaultChannelPipeline类实现fireChannelRegistered方法。Channel类实现DefaultChannelPipeline接口，fireChannelActive方法。

Channel 的 NioEventLoop 的 register0 方法，它接受一个 Runnable 参数，这个 Runnable 会在 NioEventLoop 的 register0 方法中被注册到 Channel 上。

图16-40 AbstractUnsafe 的 register 方法

register0 方法在图16-41中。

图16-41 AbstractUnsafe 的 register0 方法

ensureOpen 方法确保 Channel 是打开的，然后调用 doRegister 方法，这个方法在 AbstractNioUnsafe 的 AbstractNioChannel 类中实现，如图16-42所示。

图16-42 AbstractNioChannel 的 doRegister 方法

AbstractNioChannel 的 doRegister 方法会调用 Channel 的 ChannelPromise 方法，然后调用 ChannelPipeline 的 fireChannelRegistered 方法，这个方法在 Channel 的 ChannelPipeline 的 fireChannelActive 方法中实现。

ChannelPromise 方法

2. bind

bind 方法会调用 Channel 的 Socket 方法，如图16-43所示。

图16-43 AbstractUnsafe的bind方法

在doBind方法中，NioSocketChannel和NioServerSocketChannel都调用了图16-44的方法。

图16-44 NioSocketChannel的doBind方法

在doBind方法中，图16-45的方法。

图16-45 NioServerSocketChannel的doBind方法

在ChannelPromise和ChannelFuture的closeIfClosed方法中，Channel的

3. disconnect

disconnect方法在图16-46中。

图16-46 AbstractUnsafe的disconnect方法

4. close

在图16-47中，AbstractUnsafe的close方法调用了Runnable的run方法。

图16-47 AbstractUnsafe的close方法

在closeFuture方法和ChannelPromise方法中，调用了图16-48的方法。

JVM doClose
16-48

16-48 AbstractUnsafe close 2

ChannelPromise
ChannelPromise

ChannelOutboundBuffer close
Runnable NioEventLoop 16-49

16-49 AbstractUnsafe close 3

deregister Channel 16-50

16-50 AbstractUnsafe close 4

NioEventLoop cancel selectionKey Channel
NioEventLoop 16-51

16-51 Channel

5 write

write Channel 16-52

16-52

Channel 是 TCP Channel 的基类

1 Channel 的 TCP 连接未建立时抛出 NOT_YET_CONNECTED_EXCEPTION

2 Channel 被关闭时抛出 CLOSED_CHANNEL_EXCEPTION

ChannelPromise 的 ReferenceCountUtil.release(msg)

msg.promise 的 ReferenceCountUtil.release(msg)

6 flush

flush 是 Channel 的抽象方法

16-53

unflushed 是 tail 的抽象方法
flush0 是 flush 的抽象方法
doWrite 是 flush0 的抽象方法

16-54 doWrite 的抽象方法

unflushed - flush 的抽象方法
AbstractNioByteChannel 的 doWrite() 方法

16-55 doWrite 2

ChannelOutboundBuffer
16-56

16-56 doWrite 3

ByteBuf JDK
OP_WRITE
clearOpWrite
16-57

16-57 doWrite 4

ByteBuf
16-58

16-58 doWrite 5

ChannelOutboundBuffer remove 16-59

16-59 doWrite 6

Entry
flushed
decrementPendingOutboundBytes
incrementPendingOutboundBytes

16-60

16-60 doWrite 7

ByteBuffer false 的 DefaultSocketChannelConfig 的 doWriteBytes 方法 16-61

图 16-61 doWrite 方法

ByteBuffer readBytes() 方法返回 ByteBuffer 对象，Channel 对象是 SocketChannel 对象，ByteBuffer 对象是 SocketChannel 对象，ByteBuffer 对象是 SocketChannel 对象。

ByteBuffer 对象的 0 是 TCP 的 0，true 是 selector 的 true。

ByteBuffer 对象的 done 方法是 true。

ByteBuffer 对象的 OP_WRITE 方法是 Reactor 的 OP_WRITE 方法。

16.4.3 AbstractNioUnsafe

AbstractNioUnsafe 是 AbstractUnsafe 的子类，NIO 的 connect 和 finishConnect 方法 API。

1. connect

16-62

16-62 AbstractNioUnsafe connect 1

SocketChannel connect()

1 true

2 ACK false

3 I/O

2 NioSocketChannel selectionKey
OP_CONNECT

ChannelActive
16-63

16-63 AbstractNioUnsafe connect 2

ChannelActive
NioSocketChannel selectionKey
SelectionKey.OP_READ

16-64

16-64 AbstractNioUnsafe connect 3

1. 在 TCP 连接建立过程中，当本地主机收到来自远程主机的 SYN 报文时，会向远程主机发送 SYN-ACK 报文，此时本地主机的连接状态会从 LISTEN 变为 SYN_RECEIVED。

2. 在 TCP 连接建立过程中，当本地主机收到来自远程主机的 SYN 报文时，会向远程主机发送 SYN-ACK 报文，此时本地主机的连接状态会从 LISTEN 变为 SYN_RECEIVED。

2. finishConnect

在 TCP 连接建立过程中，当本地主机收到来自远程主机的 SYN 报文时，会向远程主机发送 SYN-ACK 报文，此时本地主机的连接状态会从 LISTEN 变为 SYN_RECEIVED。

16-65 AbstractNioUnsafe.finishConnect()

在 TCP 连接建立过程中，当本地主机收到来自远程主机的 SYN 报文时，会向远程主机发送 SYN-ACK 报文，此时本地主机的连接状态会从 LISTEN 变为 SYN_RECEIVED。

16-66 AbstractNioUnsafe.finishConnect()

在 TCP 连接建立过程中，当本地主机收到来自远程主机的 SYN 报文时，会向远程主机发送 SYN-ACK 报文，此时本地主机的连接状态会从 LISTEN 变为 SYN_RECEIVED。

- 返回 true
- 返回 false
- 抛出异常

在 TCP 连接建立过程中，当本地主机收到来自远程主机的 SYN 报文时，会向远程主机发送 SYN-ACK 报文，此时本地主机的连接状态会从 LISTEN 变为 SYN_RECEIVED。

16-67 AbstractNioUnsafe.fulfillConnectPromise()

ByteBuffer 的 read 方法返回一个 ByteBuffer，这个 ByteBuffer 包含从 SocketChannel 中读取的数据。在调用 read 方法之前，需要先调用 finishConnect 方法。图 16-68 展示了 SocketChannel 的 read 方法的调用过程。

图 16-68 AbstractNioUnsafe 的 finishConnect 方法

16.4.4 NioByteUnsafe

ByteBuffer 的 read 方法返回一个 ByteBuffer。图 16-69 展示了 ByteBuffer 的 read 方法的调用过程。

图 16-69 NioByteUnsafe 的 read 方法

ByteBuffer 的 read 方法返回一个 ByteBuffer。图 16-70 展示了 ByteBuffer 的 read 方法的调用过程。

图 16-70 SocketChannelConfig 的 API

ByteBuffer 的 read 方法返回一个 ByteBuffer。图 16-71 展示了 ByteBuffer 的 read 方法的调用过程。

图 16-71 RecvByteBufferAllocator 的 API

ByteBuffer 的 read 方法返回一个 ByteBuffer。图 16-72 展示了 ByteBuffer 的 read 方法的调用过程。

图 16-72 RecvByteBufferAllocator 的 API

64102465536
41

SIZE_TABLE16-73

16-73 SIZE_TABLE

Buffer512
512
512

AdaptiveRecvByteBufAllocator

1getSizeTableIndex(final int size)16-74

16-74

Size——

HandleImpl16-75

16-75 HandleImpl

5Buffer

record(int actualReadBytes) NioSocketChannel

```
actualReadBytes[] record[][][] [] ByteBuf[] [] []  
[] [] [] 16-76 []
```

16-76 ByteBuf

[illegible]

```

    AdaptiveRecvByteBufAllocator
}

```

□ □ □ □ □ □ □ □ □ □ □ □ □ □ □

1 Netty NIO 32K 1M
Netty Buffer

[illegible]

3 1M 1.2M
10M 10M
10M 1M

AdaptiveRecvByteBufAllocator

Handler 的 read 方法，其参数为 `byteBufCapacity`，如图 16-77 所示。Netty 的 `read` 方法，其参数为 `byteBufCapacity`，如图 16-77 所示。

图 16-77 NioByteUnsafe.read 方法

Netty 的 `ByteBuf` 类，其参数为 `byteBufCapacity`，如图 16-78 所示。

图 16-78 NioByteUnsafe.read 方法

Netty 的 `NioSocketChannel` 类，其参数为 `byteBufCapacity`，如图 16-79 所示。

图 16-79 NioByteUnsafe.read 方法

Netty 的 `javaChannel()` 方法，其参数为 `SocketChannel`，如图 16-80 所示。

图 16-80 NioByteUnsafe.read 方法

`byteBuf.writableBytes()` 方法，其参数为 `Channel`，如图 16-81 所示。

图 16-81 NioByteUnsafe.read 方法

Netty 的 `setBytes` 方法，其参数为 `byteBufCapacity`，如图 16-82 所示。

图 16-82 NioByteUnsafe.read 方法

Netty 的 `SocketChannel` 类，其参数为 `java NIO ByteBuffer`，如图 16-83 所示。Netty 的 `ByteBuf` 类，其参数为 `DK ByteBuffer`，如图 16-84 所示。Netty 的 `clear` 方法，其参数为 `position`，如图 16-85 所示。Netty 的 `index` 方法，其参数为 `index`，如图 16-86 所示。

+ByteBuffer.read(SocketChannel, ByteBuffer)
ByteBuffer

ByteBuffer

1ByteBuffer

2ByteBuffer

3ByteBuffer-1ByteBufferI/OByteBuffer

ByteBufferNettyByteBufferByteBufferByteBuffer0ByteBuffer
ByteBufferI/OByteBufferByteBufferByteBuffer0ByteBufferclose
ByteBuffer16-83

16-83 NioByteUnsafe.read8

ByteBufferChannelReadByteBuffer
ByteBufferTCPByteBuffer
ByteBufferChannelReadByteBuffer
ByteBufferChannelReadByteBuffer

ByteBufferChannelReadByteBuffer16-84

16-84 NioByteUnsafe.read9

ByteBufferTCPByteBuffer
ByteBuffer16-85

16-85 NioByteUnsafe.read10

ByteBuffer.allocate(1024).wrap(new byte[1024]).getCharSequence(0, 1024).toString().trim().length() Task
16-86

16-86 NioByteUnsafe.read 11

TCP 16
Task 16
TCP 16 selector
16-87

16-87 NioByteUnsafe.read 12

ChannelReadComplete
Hanlder record()
16-88

16-88 NioByteUnsafe.read 13

-1 I/O
16-89

16-89 NioByteUnsafe.read 14

16.5

NettyChannel——Channel
NioSocketChannelNioServerSocketChannel
“”“”

ChannelI/OUnsafe
UnsafeChannelUnsafe
Unsafe

Channel UDP
DatagramChannelEmbeddedChannel
APIdemo

17 ChannelPipeline ChannelHandler

Netty ChannelPipeline ChannelHandler Servlet Filter

Servlet Filter JEE Web Java HTTP Web

Web web.xml JSP Servlet Servlet JSP Servlet JSP Servlet Java Servlet JEE

Netty Channel Servlet Filter Channel ChannelPipeline ChannelPipeline ChannelPipeline I/O ChannelHandler ChannelHandler I/O ChannelHandler ChannelHandler ChannelHandler

ChannelPipeline ChannelHandler ChannelHandlerContext

□□□□□□□□

- ChannelPipeline□□□□
- ChannelPipeline□□□□
- ChannelHandler□□□□
- ChannelHandler□□□□

17.1 ChannelPipeline□□□□

ChannelPipeline □ ChannelHandler □ □ □ □ □ □ □
ChannelHandler□□□□□□□□□□

17.1.1 ChannelPipeline□□□□□

□17-1□□□□□□□ChannelPipeline□ChannelHandler□□□□□□□
□□□□□□□□□□□□□□□□□□□□□□□□

□17-1 ChannelPipeline□□□□□□□□□□□□

□ 1 □ □ □ □ SocketChannel read() □ □ □ □ ByteBuf □ □ □
ChannelRead□□□□I/O□□NioEventLoop□□ChannelPipeline□
fireChannelRead(Object msg) □ □ □ □ □ □ □ ByteBuf □ □ □ □
ChannelPipeline□□

□ 2 □ □ □ □ □ □ HeadHandler □ ChannelHandler1 □
ChannelHandler2.....TailHandler □ □ □ □ □ □ □ □ □ □ □ □ □ □
ChannelHandler□□□□□□□□□□□□□□□□□□□□□□□□

3 ChannelHandlerContext write TailHandler ChannelHandlerN.....ChannelHandler1 HeadHandler Future

Netty inbound outbound inbound!O TCP 17-1

inbound

1 ChannelHandlerContext.fireChannelRegistered() Channel

2 ChannelHandlerContext.fireChannelActive() TCP Channel

3 ChannelHandlerContext.fireChannelRead(Object)

4 ChannelHandlerContext.fireChannelReadComplete()

5 ChannelHandlerContext.fireExceptionCaught(Throwable)

6 ChannelHandlerContext.fireUserEventTriggered(Object)

7
ChannelHandlerContext.fireChannelWritabilityChanged()
Channel

8 ChannelHandlerContext.fireChannelInactive() TCP

Outbound I/O 17-1

outbound

1 ChannelHandlerContext.bind(SocketAddress, ChannelPromise)

2 ChannelHandlerContext.connect(SocketAddress, SocketAddress, ChannelPromise)

3 ChannelHandlerContext.write(Object, ChannelPromise)

4 ChannelHandlerContext.flush()

5 ChannelHandlerContext.read()

6
ChannelHandlerContext.disconnect(ChannelPromise)

7 ChannelHandlerContext.close(ChannelPromise) Channel
Channel

17.1.2

ChannelPipeline ChannelHandler
ChannelHandler ChannelHandler
ChannelHandler ChannelHandlerAdapter

Channel Active TCP

```
public class MyInboundHandler extends ChannelHandlerAdapter
{
    @Override
    public void channelActive(ChannelHandlerContext ctx)
    {
        System.out.println("TCP connected!");
        ctx.fireChannelActive();
    }
}
```

```
public class MyOutboundHandler extends
ChannelHandlerAdapter {
```

```

        @Override
        public void close(ChannelHandlerContext
ctx,ChannelPromise promise) {
            System.out.println("TCP closing ..");
            Object.release();
            ctx.close(promise);
        }
    }
}

```

17.1.3 管道pipeline

管道pipeline是ServerBootstrap和Bootstrap中NettyChannel中pipeline的重要组成部分。pipeline是ChannelHandler的集合。

```

pipeline = ch.pipeline();
pipeline.addLast("decoder", new MyProtocolDecoder());
pipeline.addLast("encoder", new MyProtocolEncoder());

```

ChannelHandler是MessageToMessageDecoder、ByteToMessageDecoder、ByteBuf等POJO的集合。Pipeline是ChannelHandler的集合。17-2

17-2 ChannelHandler

17.1.4 ChannelPipeline

```

ChannelPipeline[ ]ChannelHandler[ ]
[ ]
[ ]ChannelHandler[ ]ChannelPipeline
[ ]ChannelHandler[ ]

```

```
ChannelPipeline [ ] N [ ]  
ChannelPipeline[ ][ ]ChannelHandler[ ][ ]  
[ ] ChannelPipeline [ ] [ ]  
ChannelHandler[ ][ ]
```

17.2 ChannelPipeline

```

ChannelPipeline[...][ChannelHandler[...][ChannelHandler[...][ChannelHandler[...]]]]

```

17.2.1 ChannelPipeline

ChannelPipeline[17-3]

□17-3 ChannelPipeline□□□□□□

17.2.2 ChannelPipeline ChannelHandler

```

ChannelPipeline  ChannelHandler  Map  ChannelPipeline
ChannelHandler  Map  ChannelPipeline
ChannelHandler  17-4

```

图17-4 ChannelPipeline添加ChannelHandler

addBefore(ChannelHandlerInvoker invoker, String baseName, final String name, ChannelHandler handler)

图17-5

图17-5 ChannelPipeline.addBefore

ChannelPipeline

- I/O
-

ChannelPipeline

Netty synchronized

baseName DefaultChannelHandlerContext

ChannelPipeline ChannelHandler

ChannelHandlerContext

图17-6

图17-6 ChannelPipeline.context

ChannelHandler

ChannelHandler

IllegalArgumentException("Duplicate handler name: " + name)

ChannelHandler

DefaultChannelHandlerContext

图17-7

图17-7 DefaultChannelHandlerContext

图2 阻塞IO模型

fireChannelActive()方法调用了head.fireChannelActive()方法，
Channel.read()方法调用了Channel.read()方法。图17-11
所示。

图17-11 fireChannelActive()

17.2.4 ChannelPipeline outbound

ChannelPipeline的I/O模型分为outbound和inbound。outbound
Netty的pipeline模型，NIO模型
模型。

inbound模型图17-12所示。

图17-12 inbound模型

Pipeline模型I/O模型ChannelUnsafe模型
Unsafe Channel I/O Pipeline I/O
TailHandler Unsafe I/O I/O模型图17-
13所示。

图17-13 pipeline模型

TailHandler connect HeadHandler
connect图17-14所示。

图17-14 HeadHandler connect

通过HeadHandler的Unsafe.connect方法可以创建pipeline，
并返回一个Channel。

17.3 ChannelHandler接口

ChannelHandler接口是Servlet Filter的扩展，它定义了I/O操作，
并提供了ChannelHandler接口。

ChannelHandler接口定义了ChannelHandler接口，
并提供了ChannelHandler接口。

ChannelHandler接口定义了ChannelHandler接口。

- Sharable接口ChannelPipeline接口ChannelHandler接口
- Skip接口Skip接口ChannelPipeline接口ChannelHandler接口

17.3.1 ChannelHandlerAdapter接口

ChannelHandlerAdapter接口是ChannelHandler接口的实现，
它提供了ChannelHandler接口的实现。ChannelHandlerAdapter接口
ChannelHandlerAdapter接口ChannelHandlerAdapter接口
ChannelHandlerAdapter接口ChannelHandlerAdapter接口

Netty的ChannelHandlerAdapter接口是ChannelHandler接口的实现，
它提供了ChannelHandler接口的实现。ChannelHandlerAdapter接口
ChannelHandlerAdapter接口ChannelHandlerAdapter接口
ChannelHandlerAdapter接口ChannelHandlerAdapter接口
ChannelHandlerAdapter接口ChannelHandlerAdapter接口

ChannelHandlerAdapter 17-15

17-15 ChannelHandlerAdapter

17-15 @Skip ChannelHandler

17.3.2 ByteToMessageDecoder

NIO ByteBuf POJO Netty ByteToMessageDecoder

ByteToMessageDecoder void decode ChannelHandlerContext ctx, ByteBuf in, List<Object> out ByteBuf POJO

ByteToMessageDecoder TCP ByteToMessageDecoder

17.3.3 MessageToMessageDecoder

MessageToMessageDecoder Netty

SocketChannel TCP ByteBuffer ByteBuffer Java Java POJO

MessageToMessageDecoder 和 ByteToMessageDecoder 的区别

MessageToMessageDecoder 和 ByteToMessageDecoder 都是 Netty 中用于解码的类。MessageToMessageDecoder 主要用于解码 HTTP+XML 请求，而 ByteToMessageDecoder 主要用于解码 XML 请求。MessageToMessageDecoder 的 decode 方法接收一个 HttpRequest 对象，并返回一个 List<Object> 对象，其中包含一个 POJO 对象。ByteToMessageDecoder 的 decode 方法接收一个 ByteBuffer 对象，并返回一个 List<Object> 对象，其中包含一个 POJO 对象。

MessageToMessageDecoder 和 ByteToMessageDecoder 都是 Netty 中用于解码的类。MessageToMessageDecoder 主要用于解码 HTTP+XML 请求，而 ByteToMessageDecoder 主要用于解码 XML 请求。MessageToMessageDecoder 的 decode 方法接收一个 HttpRequest 对象，并返回一个 List<Object> 对象，其中包含一个 POJO 对象。ByteToMessageDecoder 的 decode 方法接收一个 ByteBuffer 对象，并返回一个 List<Object> 对象，其中包含一个 POJO 对象。

void decode(ChannelHandlerContext ctx, I msg, List<Object> out) 方法。POJO 对象。POJO 对象。ByteToMessageDecoder 类。

17.3.4 LengthFieldBasedFrameDecoder

Netty 中 TCP 连接。Netty 中 LineBasedFrameDecoder 和 DelimiterBasedFrameDecoder 类。LengthFieldBasedFrameDecoder 类。

4 字节

- 120 字节。LengthFieldBasedFrameDecoder 类。

- 0000000000000000FTP00000000000000000000
- 00000000000000
- 0000000000000000

0000000000000000LengthFieldBasedFrameDecoder00000
000000000000000000000000000000000000“0000”0000

000000000000000000000000000000000000“00”00000000000000000000000000
0017-160000

017-16 000000000000140000

0000000000000000

- lengthFieldOffset = 0
- lengthFieldLength = 2
- lengthAdjustment = 0
- initialBytesToStrip = 0

000000000000000017-170000

017-17 000000000000140000

0000ByteBuf.readableBytes()000000000000000000000000000000000000
0020000initialBytesToStrip
0002000000000000

- lengthFieldOffset = 0
- lengthFieldLength = 2
- lengthAdjustment = 0

- initialBytesToStrip = 2

17-18

17-18 12

17-18
 ByteBuf.readableBytes()

+
 17-18
 lengthAdjustment
 lengthAdjustment 17-19
 lengthAdjustment

- lengthFieldOffset = 0
- lengthFieldLength = 2
- lengthAdjustment = -2
- initialBytesToStrip = 0

17-19

lengthFieldOffset

- lengthFieldOffset = 2
- lengthFieldLength = 3
- lengthAdjustment = 0
- initialBytesToStrip = 0

17-20 初始化长度字段

1 2 2 Length 3
lengthFieldLength 3
lengthAdjustment initialBytesToStrip 0

initialBytesToStrip

- lengthFieldOffset = 1
- lengthFieldLength = 2
- lengthAdjustment = 1
- initialBytesToStrip = 3

17-21 initialBytesToStrip

HDR1 1 lengthFieldOffset 1
2 lengthFieldLength 2
lengthAdjustment 1 HDR2
HDR1 lengthAdjustment 3
13 HDR1 Length

4

TCP LengthFieldBasedFrameDecoder

```

        pipeline.addLast("frameDecoder", new
LineBasedFrameDecoder(80));

        pipeline.addLast("stringDecoder", new
StringDecoder(CharsetUtil.UTF_8));

```

在 pipeline 中添加 LineBasedFrameDecoder 和 StringDecoder 解码器，Netty 的 ByteBuf 解码器，解码器在 Netty 中被称为 ChannelHandler，解码器在 Netty 中被称为 ChannelHandler，解码器在 Netty 中被称为 ChannelHandler。

17.3.5 MessageToByteEncoder

MessageToByteEncoder 是 Netty 中一个抽象类，它实现了 ChannelHandler 接口，它的作用是将 Message 对象编码成 ByteBuf 对象。

```

MessageToByteEncoder<T> encode(ChannelHandlerContext ctx, T msg, ByteBuf out) throws Exception

```

```

public class IntegerEncoder extends
MessageToByteEncoder<Integer> {
    @Override
    public void encode(ChannelHandlerContext ctx, Integer
msg, ByteBuf out)
        throws Exception {
        out.writeInt(msg);
    }
}

```

17.3.6 MessageToMessageEncoder

POJO HTTP+XML
POJO XML HTTP
ByteBuffer

MessageToMessageEncoder void
encode(ChannelHandlerContext ctx, I msg, List<Object>
out) MessageToByteEncoder
ByteBuffer

```
public class IntegerToStringEncoder extends
MessageToMessageEncoder <Integer>
{
    @Override
    public void encode(ChannelHandlerContext ctx,
Integer message,
List<Object> out)
        throws Exception
    {
        out.add(message.toString());
    }
}
```

17.3.7 LengthFieldPrepender

Netty的LengthFieldPrepender类
ByteBuffer17-22

17-22 LengthFieldPrepender

LengthFieldPrepender类ByteBuffer2
+

LengthFieldPrependertrue
LengthFieldPrepender17-2217-23

17-23 LengthFieldPrepender

17.4 ChannelHandler

17.4.1 ChannelHandler

ByteBufferChannelChannelHandler
ChannelHandlerNetty
ChannelHandler

- ChannelPipelineChannelHandlerI/O
ChannelHandler HeadHandler TailHandler
- ChannelHandler ByteToMessageCodec MessageToMessageDecoder17-24

17-24 ChannelHandler

17-24 ChannelHandler

- ChannelHandler → Handler → Handler → Handler → Handler

□ □

17.4.2 ByteToMessageDecoder□□□□

```
ByteBuffer(ByteToMessageDecoder(ByteBufHolder.POJO))
```

channelRead 17-25

□17-25 ByteToMessageDecoder□channelRead□□

```

    msg.WriteByte(ByteBuf)
    msg.WriteByte(ByteBuf)
    msg

```

```

    cumulation
    ByteBuf
    cumulation
    ByteBuf
    ByteBuf
    cumulation

```

```
1= cumulation.readableBytes()
```

```
int n2= msg.readableBytes()
```

cumulation

17-26□□□

图17-26 ByteToMessageDecoder的expandCumulation

ByteBuffer的cumulation
ByteBuffer的cumulation
ByteBuffer的cumulation

ByteBuffer的callDecode
17-27

图17-27 ByteToMessageDecoder的callDecode

ByteBuffer的decode
17-28

图17-28 ByteToMessageDecoder的decode

pipeline
17-29

图17-29 ByteToMessageDecoder的callDecode

ChannelHandlerContext
out

1 ByteBuffer
I/O

2 ByteBuffer

17-29
Netty
readIndex
Netty

3 如果解码失败，抛出 `ByteBufDecoderException` 异常。

4 如果 `isSingleDecode` 为 `true`，则只解码一次。

17.4.3 MessageToMessageDecoder

`MessageToMessageDecoder` 是一个 `POJO`，它实现 `POJO` 接口。

它实现 `channelRead` 方法，如 17-30 所示。

图 17-30 MessageToMessageDecoder 的 `channelRead` 方法

它使用 `RecyclableArrayList` 来存储解码后的消息。它实现 `RecyclableArrayList` 接口，如 17-31 所示。

图 17-31 MessageToMessageDecoder 的 `decode` 方法

它实现 `decode` 方法，如 17-32 所示。

图 17-32 MessageToMessageDecoder 的 `decode` 方法

它使用 `ReferenceCountUtil` 的 `release` 方法来释放消息。

RecyclableArrayList

RecyclableArrayList
ChannelHandlerContext
fireChannelRead
ChannelHandler
recycle
RecyclableArrayList

17.4.4 LengthFieldBasedFrameDecoder

17-33

17-33 LengthFieldBasedFrameDecoder.decode

decode(ChannelHandlerContext ctx, ByteBuf in)
List<Object> out

decode(ChannelHandlerContext ctx, ByteBuf in)
17-34

17-34 LengthFieldBasedFrameDecoder.decode 1

discardingTooLongFrame

Math.min(bytesToDiscard, in.readableBytes())
bytesToDiscard
ByteBuf skipBytes bytesToDiscard

discardingTooLongFrame
17-35

17-35 LengthFieldBasedFrameDecoder.failIfNecessary

I/O
17-36

17-36 LengthFieldBasedFrameDecoder.decode2

lengthFieldOffset
17-37

17-37 LengthFieldBasedFrameDecoder.getUnadjustedFrameLength

6

- 1 ByteBuf.getUnsignedByte
- 2 ByteBuf.getUnsignedShort
- 3 ByteBuf.getUnsignedMedium
- 4 ByteBuf.getUnsignedInt
- 8 ByteBuf.getLong
- DecoderException

17-38
17-38

17-38 LengthFieldBasedFrameDecoder.decode3

lengthFieldEndOffset 0
CorruptedFrameException

lengthFieldEndOffset lengthAdjustment
lengthFieldEndOffset
CorruptedFrameException

ByteBuf
discardingTooLongFrame

frameLength ByteBuf
discard
discardingTooLongFrame true
failIfNecessary 17-39

17-39 LengthFieldBasedFrameDecoder decode 4

frameLength I/O

frameLength
CorruptedFrameException ByteBuf
skipBytes ByteBuf

extractFrame 17-40

17-40 LengthFieldBasedFrameDecoder extractFrame

ByteBuffer ByteBuffer
ByteBuffer ByteBuffer +
actualFrameLength

LengthField
BasedFrameDecoder
ioBuffer heapBuffer
encode
17-41

17.4.5 MessageToByteEncoder

MessageToByteEncoder POJO ByteBuffer
17-41

17-41 MessageToByteEncoder write 1

ioBuffer heapBuffer

encode 17-42

17-42 MessageToByteEncoder encode

ReferenceCountUtil release msg
ByteBuffer

- ChannelHandlerContext write
ByteBuffer
- ByteBuffer ByteBuffer
ChannelHandlerContext

ByteBuffer

17.4.6 MessageToMessageEncoder

MessageToMessageEncoder POJO POJO
XML Document XML 17-43

17-43 MessageToMessageEncoder write

RecyclableArrayList
ChannelHandler write

RecyclableArrayList
RecyclableArrayList

RecyclableArrayList POJO
17-44

17-44 POJO

17.4.7 LengthFieldPrepender

LengthFieldPrepender ByteBuffer
17-45

17-45 LengthFieldPrepender encode 1

lengthFieldLength

ByteBuf

- 1 Byte
- 2 Byte
- 3 Byte
- 4 Byte
- 8 Byte
- writeLong
- Error

ByteBuf List Object out

17.5

ChannelPipeline ChannelHandler

ChannelPipelineChannelHandler
Netty
ChannelPipelineChannelHandler

18 EventLoop EventLoopGroup

NettyNettyI/O

NettyNettyNIO NioEventLoopI/O

- Netty
- NioEventLoop

18.1 Netty

NettyReactorNIOReactorReactor

Reactor

18.1.1 Reactor

ReactorNIONION

- NIOTCP
- NIOTCP

- 阻塞IO模型
- 非阻塞IO模型

Reactor模型18-1

图18-1 Reactor模型

Reactor模型是一种IO模型，它通过一个或多个线程（Reactor线程）来处理IO请求。Reactor线程通过监听（Accept）新的连接，并将连接分派（Dispatch）给相应的Handler（处理器）来处理。Handler通过阻塞IO（如NIO）来读取或写入数据。Reactor模型的优点是它可以避免多线程同时处理IO请求导致的资源浪费，同时它也可以避免单线程处理IO请求导致的性能瓶颈。

Reactor模型的优点是它可以避免多线程同时处理IO请求导致的资源浪费，同时它也可以避免单线程处理IO请求导致的性能瓶颈。

- NIO模型可以避免多线程同时处理IO请求导致的资源浪费，同时它也可以避免单线程处理IO请求导致的性能瓶颈。
- NIO模型可以避免多线程同时处理IO请求导致的资源浪费，同时它也可以避免单线程处理IO请求导致的性能瓶颈。
- NIO模型可以避免多线程同时处理IO请求导致的资源浪费，同时它也可以避免单线程处理IO请求导致的性能瓶颈。

Reactor模型是一种IO模型，它通过一个或多个线程（Reactor线程）来处理IO请求。Reactor线程通过监听（Accept）新的连接，并将连接分派（Dispatch）给相应的Handler（处理器）来处理。Handler通过阻塞IO（如NIO）来读取或写入数据。

18.1.2 Reactor模型

Reactor模型是一种IO模型，它通过一个或多个线程（Reactor线程）来处理IO请求。Reactor线程通过监听（Accept）新的连接，并将连接分派（Dispatch）给相应的Handler（处理器）来处理。Handler通过阻塞IO（如NIO）来读取或写入数据。

Reactor模型

- 模型NIO——Acceptor模型——TCP模型
- I/O——模型NIO模型——JDK模型
- NIO模型——NIO模型

Reactor模型——NIO模型

Acceptor模型——Reactor模型

18.1.3 Reactor模型

Reactor模型——NIO模型

NIO模型——Acceptor模型——TCP模型

SocketChannel模型——I/O模型——sub reactor模型——I/O模型

SocketChannel模型——Acceptor模型——NIO模型

subReactor模型——I/O模型——I/O模型

18-3

NIO模型——demo模型

18.1.4 Netty

```
Netty[0][0][0][0][0][0][0][0][0][0][0][0][0][0][0][0][0][0][0][0][0][0]
Netty[0][0][0][0]Reactor[0][0][0][0][0][0][0][0]Reactor[0][0][0][0]
```

18-4 Netty

18-4 Netty

18-5 Netty

18-5 Netty

```

    NioEventLoopGroup
ReactorTCP
TaskTask

```

Netty

1 TCP Channel

2 ChannelPipeline

Netty I/O Reactor

1 ChannelPipeline

ChannelPipeline

Task

4 Task

Netty的Reactor模型是事件驱动模型，它通过注册感兴趣的事件，当事件发生时，Netty会通知相应的处理器来处理该事件。

Netty的Reactor模型是事件驱动模型，它通过注册感兴趣的事件，当事件发生时，Netty会通知相应的处理器来处理该事件。CPU通过轮询的方式，不断地检查是否有事件发生，一旦有事件发生，CPU就会调用相应的处理器来处理该事件。Netty的Reactor模型是事件驱动模型，它通过注册感兴趣的事件，当事件发生时，Netty会通知相应的处理器来处理该事件。

18-6

18-6 Netty Reactor模型

Netty的NioEventLoop模型是事件驱动模型，它通过注册感兴趣的事件，当事件发生时，Netty会通知相应的处理器来处理该事件。fireChannelRead (Object msg) 是Netty的NioEventLoop模型中的一个方法，它用于处理接收到的消息。Netty的NioEventLoop模型是事件驱动模型，它通过注册感兴趣的事件，当事件发生时，Netty会通知相应的处理器来处理该事件。

18.1.5

Netty的NioEventLoop模型是事件驱动模型，它通过注册感兴趣的事件，当事件发生时，Netty会通知相应的处理器来处理该事件。

1. Netty的NioEventLoopGroup模型是事件驱动模型，它通过注册感兴趣的事件，当事件发生时，Netty会通知相应的处理器来处理该事件。NIO Acceptor是Netty的NioEventLoopGroup模型中的一个组件，它用于接收来自客户端的连接。

2. Netty的ChannelHandler模型是事件驱动模型，它通过注册感兴趣的事件，当事件发生时，Netty会通知相应的处理器来处理该事件。POJO是Netty的ChannelHandler模型中的一个组件，它用于处理接收到的消息。

3. Netty的NIO模型是事件驱动模型，它通过注册感兴趣的事件，当事件发生时，Netty会通知相应的处理器来处理该事件。

第4步：在NIO中，我们使用Selector来管理多个Channel。Selector是一个线程，它负责监听多个Channel的I/O事件。当某个Channel有事件发生时，Selector会通知主线程，主线程再对相应的Channel进行操作。

第5步：在NIO中，我们使用POJO来封装数据。POJO是一个普通的Java对象，它不包含任何特殊的方法或属性。Task是一个接口，它定义了任务的基本操作。NIO中，我们使用Task来封装任务，并使用Selector来管理任务。

在NIO中，我们使用Selector来管理多个Channel。Selector是一个线程，它负责监听多个Channel的I/O事件。当某个Channel有事件发生时，Selector会通知主线程，主线程再对相应的Channel进行操作。

- $QPS = \frac{\text{请求数}}{\text{时间}} \times \text{并发数}$
- $QPS = 1000 / \text{时间} \times \text{并发数}$

在NIO中，我们使用Selector来管理多个Channel。Selector是一个线程，它负责监听多个Channel的I/O事件。当某个Channel有事件发生时，Selector会通知主线程，主线程再对相应的Channel进行操作。

18.2 NioEventLoop

18.2.1 NioEventLoop

Netty的NioEventLoop是一个线程，它负责监听多个Channel的I/O事件。当某个Channel有事件发生时，NioEventLoop会通知主线程，主线程再对相应的Channel进行操作。

- Task接口定义了NioEventLoop的execute(Runnable task)方法。Netty使用Task接口来封装任务，并使用NioEventLoop来管理任务。Task接口定义了execute()方法，用于执行任务。NioEventLoop使用Task接口来管理任务，并使用execute()方法来执行任务。
- NioEventLoop的schedule(Runnable command, long delay, TimeUnit unit)方法用于调度任务。该方法用于在指定的时间后执行指定的任务。

Netty的NioEventLoop是一个线程，它负责监听多个Channel的I/O事件。当某个Channel有事件发生时，NioEventLoop会通知主线程，主线程再对相应的Channel进行操作。Runnable接口定义了run()方法，用于执行任务。18-7

图18-7 NioEventLoop的组成

图中展示了EventLoop和EventExecutorGroup的组成。EventLoop包含ScheduledExecutorService和NioEventLoop。EventExecutorGroup包含多个EventLoop。图中还展示了NioEventLoop的组成，包括Selector、Channel、ChannelHandlerContext、ChannelHandler等。

18.2.2 NioEventLoop的组成

图中展示了NioEventLoop的组成。NioEventLoop包含Selector、Channel、ChannelHandlerContext、ChannelHandler等。图中还展示了Netty的Reactor模型，其中NioEventLoop是Reactor的核心组件。

图18-8 NioEventLoop的组成

18.2.3 NioEventLoop

图中展示了NioEventLoop的组成。NioEventLoop包含Selector、Channel、ChannelHandlerContext、ChannelHandler等。图中还展示了NioEventLoop的组成，包括Selector、Channel、ChannelHandlerContext、ChannelHandler等。

图18-9 NioEventLoop的Selector

Selector的open()方法用于创建Selector。Selector的selectedKeys()方法用于获取已经就绪的通道。图中还展示了io.netty.noKeySetOptimization的优化选项，用于提高Selector的性能。

Selector的组成如图18-10所示。

图18-10 Selector的组成

selectedKeys 是 provider.openSelector() 返回的
一个 Selector 对象

Selector 的 selectedKeys 是
public SelectedKeys 返回的。Netty 的
selectedKeys 是 selectedKeySet 的 JDK 的 selectedKeys 的

Selector 的 run 方法 18-11

18-11 NioEventLoop 的 run 方法

for 循环中 NioEventLoop 的
NIO 的

wakenUp 是 false 的 wake up 的
oldWakenUp 的 hasTasks() 的
selectNow() 的 select 的 Channel 的
18-12

18-12 NioEventLoop 的 selectNow() 方法

Selector 的 selectNow() 的 Selector 的
Channel 的 Channel 的 0 的
Selector 的 wakeup 的 selector.wakeup() 的

run 的 select() 的
Selector 的 Channel 的 18-13

18-13 NioEventLoop 的 select() 方法

delayNanos() NioEventLoop
select

0.5 selector.selectNow()
selectCnt 1

select select select
selectCnt 1

Select

- Channel selectedKeys 0
- oldWakeup true
- wakeup
-

Selector wakeup
JDK epoll bug Selector I/O
100% JDK7 bug Netty bug

Bug-id=6403933 Selector 18-14

18-14 JDK Selector CPU 100% bug

Bug

1 Selector select

2. select 阻塞

3. 100ms 轮询 NIO 的 JDK NIO 的 `epoll()` bug

Selector 阻塞 Selector 阻塞 18-15

18-15 Selector

`inEventLoop()` 阻塞 `rebuildSelector` 阻塞 Selector 阻塞 `rebuildSelector` 阻塞 Task `NioEventLoop` 阻塞 `NioEventLoop` 阻塞

`openSelector` 阻塞 Selector 阻塞 Selector 阻塞 `SocketChannel` 阻塞 Selector 阻塞 Selector 阻塞 Selector 阻塞

18-16

18-16 `SocketChannel` 阻塞 Selector

阻塞 Selector 阻塞 Selector 阻塞 I/O CPU 100%

阻塞 `SocketChannel` I/O 阻塞 18-17

18-17 I/O

selectedKeys 调用 processSelectedKeysPlain 方法
processSelectedKeysPlain 方法

18-18 SelectionKey 方法

SelectionKey 方法
SelectionKey 方法
SelectionKey 方法
18-19

18-19 SocketChannel 方法

SocketChannel 方法
AbstractNioChannel 方法
NioServerSocketChannel 方法
NioSocketChannel 方法
I/O 方法
NioTask 方法
processSelectedKey 方法
Netty 方法
NioTask 方法
Task 方法

I/O 方法 18-20

18-20 方法

NioServerSocketChannel 方法
NioSocketChannel 方法
Unsafe 方法
Unsafe 方法
close 方法

方法 18-21

18-21 方法

Unsafe.readUnsafe
NioServerSocketChannelTCP18-22

18-22 NioServerSocketChannel

NioSocketChannel SocketChannel
ByteBuffer18-23

18-23 NioSocketChannel

flush
18-24

18-24 flush

18-25

18-25

finishConnect
SelectionKey.OP_CONNECT

NioEventLoopI/OTask
18-26

18-26 I/O

NioEventLoopI/OI/O
CPU
NettyI/OI/OTask
I/O50%

Task 阻塞等待 I/O 操作完成后再继续执行 runAllTasks 方法
图 18-27

图 18-27 阻塞 I/O

阻塞 I/O 操作会阻塞整个线程，导致其他任务无法执行。Task Queue 可以解决这个问题。
图 18-28

图 18-28 非阻塞 I/O 和 Task Queue

Task Queue 可以处理阻塞 I/O 操作，确保其他任务可以继续执行。
图 18-29

图 18-29 非阻塞 I/O 和 Task Queue

阻塞 I/O 操作会阻塞整个线程，导致其他任务无法执行。Task Queue 可以解决这个问题。
图 18-30

图 18-30 NioEventLoop 阻塞 I/O

Channel 的 Unsafe.close() 方法会关闭 ChannelPipeline 和 ChannelHandler。
图 18-31

18.3 小结

NettyNioEventLoop
Netty

NettyReactor

第19章 Future和Promise

本章主要介绍Netty中的Future和Promise，以及它们的使用。Future和Promise是异步编程中的两个重要概念，它们可以帮助我们处理异步操作的结果。Netty Future和Promise的使用非常灵活，可以满足各种异步编程的需求。

本章主要介绍

- Future
- Future
- Promise
- Promise

19.1 Future

Future是JDK中`java.util.concurrent.Future`接口，它定义了异步操作的结果。API如图19-1所示。

图19-1 JDK Future API

`get()`方法用于获取异步操作的结果。如果操作已经完成，则返回结果；否则，抛出`TimeoutException`。

`isDone()`方法用于判断异步操作是否已经完成。如果已经完成，则返回`true`；否则，返回`false`。

`cancel()`方法用于取消异步操作。如果操作已经启动，则返回`true`；否则，返回`false`。

ChannelFuture

NettyFutureI/OChannelFutureChannel

API19-1

19-1 ChannelFuture

NettyI/OI/OBIOChannelFuture

ChannelFutureuncompletedcompletedI/OChannelFutureuncompleted——I/OChannelFuturecompleted

-
-
-

ChannelFuture19-2

19-2 ChannelFuture

ChannelFutureAPII/O

19-3

图19-3 ChannelFuture的用法

Netty的I/O模型中，ChannelFuture的用法如下：

ChannelFuture的GenericFutureListener的remove方法，可以移除GenericFutureListener。

GenericFutureListener的图19-4如下：

图19-4 GenericFutureListener的用法

I/O模型中，ChannelFuture的GenericFutureListener的operationComplete方法，可以移除ChannelFuture。

GenericFutureListener的ChannelFuture的get方法，可以移除I/O模型中的I/O操作。GenericFutureListener的ChannelFuture的get方法，可以移除I/O操作。

ChannelHandler的ChannelFuture的await()方法，可以移除I/O模型中的I/O操作。ChannelHandler的ChannelFuture的await()方法，可以移除I/O操作。

图19-5如下：

图19-5 ChannelFuture的用法

调用`await()`阻塞等待I/O操作完成并调用`notify()`唤醒等待的线程。I/O操作完成后调用`getNow()`返回结果。如果操作失败，抛出`ExecutionException`。

图19-10 阻塞等待I/O操作

图19-10 阻塞等待I/O操作

调用`await(long timeout, TimeUnit unit)`阻塞等待I/O操作完成，如果超时，抛出`TimeoutException`。调用`get()`返回结果。如果操作失败，抛出`ExecutionException`。

`ChannelFuture`是`Future`的子接口。它提供了`channel()`方法，返回相关的`Channel`。图19-11展示了`AbstractFuture`的实现。

图19-11 AbstractFuture的实现

19.3 Promise

`Promise`是`Future`的子接口。它提供了`set()`方法，用于设置结果。图19-12展示了`Netty`中的`Promise`实现。

图19-12 Netty中的Promise实现

图19-13 Promise的实现

图19-13 Promise的实现

Netty 的 I/O 模型中，`Promise` 是 `ChannelHandler` 的 `Context.write(Object object)` 方法返回的 `ChannelPromise` 对象。`19-14`

`19-14` I/O 模型中的 `Promise`

在 I/O 模型中，`Promise` 是 `19-15`

`19-15` I/O 模型中的 `tryFailure` 方法

19.4 Promise

19.4.1 Promise

在 I/O 模型中，`Promise` 是 `19-16`

`19-16` Promise

`Promise` 是 `19-16`

19.4.2 DefaultPromise

`setSuccess` 方法 `19-17`

`19-17` `DefaultPromise` 的 `setSuccess` 方法

`setSuccess0` 方法 `19-17`

setSuccess0 19-18

19-18 DefaultPromise setSuccess0

Promise
19-18

I/O Promise
19-18

19-18

result notify
result SUCCESS
result

I/O notifyAll
notifyAll wait

setSuccess0 await 19-19

19-19 DefaultPromise await

Promise
Promise isDone
Effective Java
243 wait notify

非阻塞I/O的Promise的await和sync方法，它们都返回一个Future对象，这个Future对象实现了java.lang.Object.wait()方法，因此可以通过调用wait()方法来等待I/O操作完成。setSuccess、trySuccess、setFailure、tryFailure等方法用于设置Future对象的状态。

19.5 总结

本章介绍了Future和Promise的概念，以及它们在Netty中的应用。Future是一个接口，它表示一个异步计算的结果。Promise是一个接口，它表示一个异步计算的结果，并且可以通过Future-Listener来监听它的状态。JDK提供了Future和Promise的实现，Netty也提供了自己的实现。

Future和Promise的await和sync方法，它们都返回一个Future对象，这个Future对象实现了java.lang.Object.wait()方法，因此可以通过调用wait()方法来等待I/O操作完成。

📖📖📖📖📖📖📖 **Netty**📖📖📖📖

📖20📖 Java📖📖📖📖📖Netty📖📖📖📖

📖21📖 Netty📖📖📖📖

📖22📖 Netty📖📖📖📖

📖23📖 Netty📖📖📖📖

第20章 Java网络编程之Netty

本章主要介绍Netty框架，Netty是一个基于NIO的异步事件驱动的网络框架，它可以帮助开发人员快速开发高性能的网络应用。Netty框架的引入，使得网络编程变得更加简单和高效。

本章主要介绍

- Java网络编程
- Netty框架

20.1 Java网络编程

20.1.1 网络编程概述

网络编程是指通过计算机网络进行数据通信和交换的过程。在Java中，网络编程通常涉及到Socket、ServerSocket、DatagramSocket等类。网络编程的核心是数据的发送和接收，这涉及到I/O操作。CPU在等待I/O操作完成时会处于空闲状态，这被称为I/O wait。

Java网络编程主要分为同步和异步两种。同步网络编程是指程序在发送或接收数据时必须等待I/O操作完成，这会导致程序在等待期间处于空闲状态。异步网络编程是指程序在发送或接收数据时不需要等待I/O操作完成，这可以提高程序的效率。Java提供了多种网络编程模型，包括阻塞式、非阻塞式、多路复用等。

20.1.2 Java网络编程

- assign 變數的資料型態與值
- store 變數的資料型態與值
- write 變數的資料型態與值

3 Java

Java 是一種面向物件的程式語言，它是由 Sun 公司所開發的。Java 的語法與 C++ 類似，但它具有更強的封裝性與安全性。Java 的執行環境是 JVM（Java Virtual Machine），它可以在不同的作業系統上運行。

Java 的執行環境是 JVM（Java Virtual Machine），它可以在不同的作業系統上運行。Java 的執行環境是 JVM（Java Virtual Machine），它可以在不同的作業系統上運行。

Java 的執行環境是 JVM（Java Virtual Machine），它可以在不同的作業系統上運行。

1. Java 的執行環境是 JVM（Java Virtual Machine），它可以在不同的作業系統上運行。

2. Java 的執行環境是 JVM（Java Virtual Machine），它可以在不同的作業系統上運行。

3. Java 的執行環境是 JVM（Java Virtual Machine），它可以在不同的作業系統上運行。

Java 的執行環境是 JVM（Java Virtual Machine），它可以在不同的作業系統上運行。Java 的執行環境是 JVM（Java Virtual Machine），它可以在不同的作業系統上運行。Java 的執行環境是 JVM（Java Virtual Machine），它可以在不同的作業系統上運行。

20.2 Netty

20.2.1 □□□□□□□□□□

```

    synchronized
    {
        // ...
    }

```

NettyNetty

ServerBootstrapOption 20-2

□20-2 □□□□□□□□

```

    ServerBootstrap ServerSocketChannel
    Socket

```

[LinkedListHashMap](#) API DOC
20-3

20-3 LinkedHashMap API

```

    ServerBootstrap bootstrap(
        ChannelOptions options,
        ChannelOption<T> option, T value) {
        ...
    }
}

```

Netty

11

20.2.2

Netty

ForkJoinTask
 20-4

□20-4 □□□□□

```

    status=0
    wait()
    status=0
    
```

```

1wait

```

```
synchronized(this)
{
    While(condition)
        Object.wait;
    .....
}
```

```

2waitwaitwait
notifyAll()

```

notify()和notifyAll()方法只能由持有该对象的锁的线程调用。

3. 在调用notify()或notifyAll()之前，必须确保当前线程持有该对象的锁。
在调用notifyAll()之前，必须确保当前线程持有该对象的锁。
在调用notify()之前，必须确保当前线程持有该对象的锁。

在调用notify()或notifyAll()之前，必须确保当前线程持有该对象的锁。
在调用notifyAll()之前，必须确保当前线程持有该对象的锁。
在调用notify()之前，必须确保当前线程持有该对象的锁。
在调用notifyAll()之前，必须确保当前线程持有该对象的锁。
在调用notify()之前，必须确保当前线程持有该对象的锁。

```
int size = 0;
public synchronized void increase()
{
    size++;
}
public int current()
{
    Return size;
}
```

20.2.3 volatile

volatile是Java中的一个关键字，用于修饰变量。volatile变量保证在多线程环境下，对变量的读写操作是可见的。volatile变量只能用于修饰变量，不能用于修饰方法或代码块。

volatileNettyvolatile

NioEventLoopI/OioRatioint

volatilevolatileNetty

volatileJavaJavavolatile

volatile

- volatile
- 20-4

20-4

3s

```
if (!stop)
While(true)
.....
```

stop 方法中，我们使用 volatile 来保证 stop 方法的可见性。

图 20-5 volatile 保证可见性

在图 20-5 中，我们使用 volatile 来保证 stop 方法的可见性。

- main 方法中，我们使用 stop 方法来停止 workThread。
- workThread 方法中，我们使用 stop 方法来停止。

在图 20-5 中，我们使用 volatile 来保证 stop 方法的可见性。在图 20-5 中，我们使用 volatile 来保证 stop 方法的可见性。

在图 20-5 中，我们使用 volatile 来保证 stop 方法的可见性。在图 20-5 中，我们使用 volatile 来保证 stop 方法的可见性。

在图 20-5 中，我们使用 volatile 来保证 stop 方法的可见性。在图 20-5 中，我们使用 volatile 来保证 stop 方法的可见性。

图 20-6 volatile 与 NioEventLoop

在图 20-6 中，我们使用 volatile 来保证 stop 方法的可见性。在图 20-6 中，我们使用 volatile 来保证 stop 方法的可见性。

图 20-7 volatile 与 NioEventLoop

而NioEventLoop中所有IO操作都是阻塞的，
所以volatile和synchronized在Netty中
很少使用。

Netty中volatile使用非常少，
Netty volatile使用非常少。

20.2.4 CAS

CAS（Compare and Swap）是一种非阻塞的同步方法，
它通过比较内存中的值与预期值，如果相等则更新为新的值，
否则不更新。CAS操作通常用于实现无锁的数据结构，
如计数器、队列等。

Java中CAS操作由sun.misc.Unsafe类提供，
IA64、X86、Sparc-TSO、case、ARM、
PowerPC等处理器都支持ldrex/strex指令。

JDK1.5中引入了CAS操作，sun.misc.Unsafe类
提供了compareAndSwapInt()、compareAndSwapLong()
等方法。sun.misc.Unsafe类在JDK中
AtomicInteger。

Netty中CAS操作使用非常少。

ChannelOutboundBuffer中CAS操作使用非常少。
20-8

图20-8 伪代码

```
volatile总pendingSize
// 总pendingSize的更新器
AtomicIntegerFieldUpdater
WTOTAL_PENDING_SIZE_UPDATER总pendingSize
// 总pendingSize的更新器
AtomicIntegerFieldUpdaterAPI图20-9
```

图20-9 AtomicIntegerFieldUpdater Java DOC

```
APIvolatileint
compareAndSetset
```

```
write
writeI/O
volatile
图20-10
```

图20-10 伪代码

```
CAS TOTAL_
PENDING_SIZE_UPDATER.compareAndSet(this, oldValue,
newWriteBufferSize)
oldValue
```

```
oldValue = totalPendingSize;
```

newWriteBufferSize = oldValue + size;

AtomicInteger compareAndSet
CAS

Java Atomic
Netty int long boolean

20.2.5

JDK1.5 java.util.concurrent
Java

4

- Executor Framework Timer
- List Queue Map Set
- ReadWriteLock
- AtomicInteger

Task Runnable/Callable
wait notify

Netty

NettyNioEventLoopI/O
I/O
Task20-11

20-11

ConcurrentLinkedQueueAPI20-12

20-12 ConcurrentLinkedQueue

DOC
20-13

20-13 ConcurrentLinkedQueueTask

20-14

20-14 ConcurrentLinkedQueueTask

JDKCASvolatileReadWriteLock
20-15

NettySingleThreadEventExecutor

20-15

图20-15 SingleThreadEventExecutor示意图

该图展示了SingleThreadEventExecutor的内部结构。它包含一个名为`run`的方法，该方法调用了`Task`接口。图中还显示了`NioEventLoop`和`run`方法的调用关系。图20-16

图20-16 SingleThreadEventExecutor示意图

该图展示了SingleThreadEventExecutor的内部结构。它包含一个名为`run`的方法，该方法调用了`Task`接口。图中还显示了`NioEventLoop`和`run`方法的调用关系。图20-17

图20-17 该I/O线程的Task

图20-18展示了该I/O线程的Task。图中显示了Task的调用关系。

图20-18 该I/O线程的Task

Netty的JDK版本。图中显示了JDK版本的调用关系。

20.2.6 该I/O线程

JDK1.5版本。图中显示了JDK1.5版本的调用关系。

该JDK1.6版本的JVM。图中显示了JDK1.6版本的JVM的调用关系。

Netty的HashedWheelTimer。图中显示了Netty的HashedWheelTimer的调用关系。

ReentrantReadWriteLock 20-19 wheel newTimeout

20-19 Read Lock

20-20

20-20 Write Lock

-
-
- ReentrantReadWriteLock
- false tryLock
- finally tryLock

20.2.7

Netty 的 ChannelPipeline 是 Netty 的核心，它负责管理 Channel 的读写操作。Netty 的 ChannelPipeline 是一个链式结构，每个 Channel 都有一个 ChannelPipeline，每个 ChannelPipeline 由一个或多个 ChannelHandler 组成。ChannelHandler 负责处理 Channel 的读写操作，包括数据的接收、解码、编码、发送等操作。

Netty 的 ChannelPipeline 的 API 文档在 20-21 页。Netty 的 ChannelPipeline 的 API 文档在 20-21 页。

20-21 ChannelPipeline 的 API

ChannelPipeline 的 API 文档在 20-21 页。ChannelPipeline 的 API 文档在 20-21 页。

20.2.8 ChannelPipeline 的 API

ChannelPipeline 的 API 文档在 20-21 页。ChannelPipeline 的 API 文档在 20-21 页。

Netty 的 ChannelPipeline 的 API 文档在 20-21 页。Netty 的 ChannelPipeline 的 API 文档在 20-21 页。

20-22 ChannelPipeline 的 API

20.3 ChannelPipeline 的 API

ChannelPipeline 的 API 文档在 20-21 页。ChannelPipeline 的 API 文档在 20-21 页。

ChannelPipeline 的 API 文档在 20-21 页。ChannelPipeline 的 API 文档在 20-21 页。

[illegible]

21.1.3 `Service ChannelHandler`

CMPP

[illegible]

```

NettyNetty

```

21.2 □□□□□□□□

21.2.1 □□□

[illegible]

- [illegible]

--	--	--	--	--	--	--

- 网络吞吐能力
- 网络IOPS
- 网络延迟

网络吞吐能力是指单位时间内网络能够处理的数据量，通常用每秒传输的数据量来衡量。

“网络吞吐能力”通常用Netty来衡量。

1. 网络I/O模型Reactor模型，网络I/O模型，网络I/O模型。

2. TCP模型，网络I/O模型。

3. ByteBuf模型，ByteBuf模型。

4. 网络I/O模型TCP模型，网络I/O模型。

5. 网络I/O模型，网络I/O模型。

6. 网络I/O模型，网络I/O模型。

7. 网络I/O模型，网络I/O模型，CPU模型。

8. 网络I/O模型，网络I/O模型，GC模型，CPU模型。

NettyNettyNIO
Netty21-221-321-421-5

21-2 64128

21-3 2561K

21-4 64128

21-5 2561K

21.2.2

Netty

1

HangTCP

Netty

- `Handler`
- `Handler`

Netty

2

Netty

- `ByteBuf`
- `ByteBuf`
- `ByteBuf`

`AbstractReferenceCountedByteBuf` 21-6 21-7

21-6

21-7

`ByteBuf` 21-8

21-8

NettyNIOHTTPThrift
FTPNettyNetty
Netty

NettyNettyHTTPDubbo
RocketMQ

21.3

NettyNetty
NettyNetty
Netty

第22章 Netty入门

Tomcat Web

Hadoop MapReduce T

Hadoop RPC Netty

- Netty
- Netty
- Netty

22.1 Netty入门

22.1.1

22-1

图22-1 部署架构图

部署架构图

1. 部署架构图

2. 部署架构图RPC

3. 部署架构图

4. 部署架构图

22.1.2 部署架构图Dubbo

部署架构图DubboTomcat

1. 部署架构图RPC

2. 部署架构图URLF5

3. 部署架构图

4. 部署架构图

Dubbo

1 Netty+ ——Dubbo
RMI+HTTP+XML

2 Zookeeper
Failover F5

3

4

Dubbo 22-2

22-2 Dubbo

22.1.3 Dubbo

Dubbo Zookeeper 22-3

22-3 Dubbo

- Provider
- Consumer
- Registry
- Monitor
- Container

- 消息的发送和接收
- 消息的序列化与反序列化
- 消息的压缩与解压
- 消息的加密与解密
- 消息的鉴权与授权
- 消息的日志记录

3. 消息的传输

- 消息的传输方式
- 消息的传输协议

4. 消息的接收

- 消息的接收方式

22-4 Dubbo消息的接收

22.1.4 Netty与Dubbo消息的接收

Dubbo的RPC消息的接收Netty+Dubbo消息的接收RPC消息的接收

- 消息的接收方式
- 消息的接收协议
- 消息的接收方式
- JDK消息的接收方式

消息的RPC消息的接收

- 序列化与反序列化
- 序列化与反序列化
- 序列化与反序列化

Dubbo RCP 序列化 Dubbo 序列化 Hessian 序列化 I/O

Dubbo 序列化 22-5

22-5 Dubbo 序列化

Dubbo RPC Dubbo encode POJO Dubbo Netty Client Netty Server NioSocketChannel ByteBuffer Dubbo RPC Dubbo 22-6

22-6 Dubbo 序列化

22.1.5 Dubbo Netty

Netty Dubbo RPC NIO

1 NIO

2 NIO

3

4

5

6 Dubbo Handler

Netty transport netty
22-7

22-7 Netty

Dubbo TCP NIO Dubbo RPC
Netty Netty NIO
Dubbo Mina grizzly NIO
NIO

NettyClient
22-8

22-8 Netty

Dubbo Netty

1 ChannelPipeline Dubbo

2 ChannelPipeline Dubbo

3 ChannelPipeline Dubbo Handler

Dubbo Netty Handler 22-9

22-9 Dubbo ChannelHandler

```

NettyHandler  [] [] Dubbo  [] [] [] ChannelHandler [] [] []
ChannelHandler[]Dubbo[]Filter[]RPC[] [] [] []22-10 [] [] []

```

22-10 DubboFilter

Dubbo□Netty□□□□□□□□

- □□□□□□□□□□□□□□
- □□□□□□□□□□

□□□□□□□□22-11□□□

22-11 Dubbo Netty

demo

```

    DubboNetty
    ScheduledThreadPoolExecutor
    HeartBeatTask
    HeartbeatHandler

```

□□□□□□□□22-12□□□

□22-12 □□Dubbo□□

22-13

□22-13 □□Dubbo□□

[illegible]

Task 22-14

22-14 Dubbo Task

Channel 22-15

22-15 Dubbo

22.2 Netty

Apache Avro Hadoop Doug Cutting Hadoop Writable
——

Avro Schema Schema Schema Schema
Schema Schema Schema Schema
Thrift Avro Hadoop RPC

Avro RCP

- Jetty HTTP Server
- Netty Netty Server

Avro

1 Netty HTTP

2. Avro Protocol Buffer JSON Hessian Java

Avro

1. Avro

2. Avro Hang

3. Avro RPC MQ TOP

4. Avro Hadoop Thrift Protocol Buffer IDL

5. Avro Protocol Buffer Java JSON

Netty Avro RPC Dubbo NIO Avro

22.3 Netty

MMORPG

- 服务器端开发语言PC端开发语言
- 服务器端开发语言JavaJS++PHP
- 数据库
- 中间件
- 网络
- 操作系统

服务器端开发语言PC端开发语言

Netty

22.3.1 网络编程

网络编程22-16

22-16 网络编程

网络编程22-17

22-17 网络编程

网络编程

①网络编程

②网络编程

③网络/网络

④网络编程

⑤网络/网络

⑥□□□□□□□□

⑦□□/□□□

⑧□□□□ID□□□□□□

⑨□□/□□□□□□□□□□

□□□□□□□□22-18□□□□□□□□

□22-18 □□□□□□□□

①□□□□□□□□□□

②□□□□□□

③□□□□□□□

④□□□□□□□□□

⑤□□□□□□□

⑥□□□□□□□□□□

⑦□□□□□□□

⑧□□ID□□□□□□□□□□

⑨□□□□□□□□□□□□

⑩□□□□ID□□□□□□ID□□□□□□

□□□□□□□□□□□□□□□□

□□□□□□□□22-19□□

□22-19 □□□□□□□□

① □□□□□□□□

② 姓名ID

③ □□□□□□□□

[illegible]

⑤ □□□□□□□

⑥ □□□□□□□□

(7)

⑧ □□□□□□□□□□

□□□□□□□□□□□□□□22-20□□

□22-20 □□□□□□□□

22.3.2 Netty□□□□□□□□

Netty

1
Netty NIO

2

3 TCP

4 SSL

5

6 GC

Netty 22-21

22-21 Netty

22.4

Netty NIO RPC Netty

RPC Netty Netty

第23章 Netty入门

本章主要介绍Netty入门

本章主要介绍

- 简介
- 安装
- 快速入门
- Road Map

23.1 简介

Netty是一个高性能的异步事件驱动的网络应用程序框架，使用纯Java实现，旨在帮助开发人员快速编写出高性能的网络应用程序。

Netty是一个高性能的异步事件驱动的网络应用程序框架，使用纯Java实现，旨在帮助开发人员快速编写出高性能的网络应用程序。Netty + 其他框架（如Dubbo、RocketMQ、Hadoop、Avro等）可以构建出高性能的分布式系统。

Netty基于JDK7的NIO2.0实现，支持Socket、UDP、TCP、HTTP、HTTPS、WebSocket、WebRTC等协议。Netty的Mina实现基于NIO1.4实现。

Netty2.3版本支持NIO2.0实现，Java8版本支持Netty3.0版本。Netty3.0版本支持NIO2.0实现。

23.2 简介

目前JDK8仍然是ORACLE目前唯一支持JDK7的发行版，JDK6的deadline是Netty 5.X，目前仍然是JDK7的发行版，JDK7的发行版是Netty 5.X，目前仍然是NIO2.0的发行版，AIO的发行版

目前仍然是Netty 5.0的发行版

23.3 目前情况

Netty目前仍然是API，目前仍然是Bug，目前仍然是Netty，目前仍然是

23.4 Road Map

Netty目前仍然是

- Bug目前仍然是
- 目前仍然是

目前仍然是Netty 4.X，目前仍然是23-1

23-1 Netty 4.X目前仍然是

2013年12月22日，Netty 5.0.0.Alpha1发布，2014年Netty 5.0.0发布，5.0.0发布，Final发布，2014年Netty 5.0.0发布，Netty 5.0.0发布

23.5 目前

NettyNIO
NettyNetty
Netty

□□ **Netty**□□□□□
